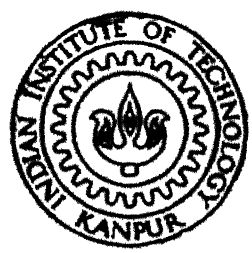
Parallel Algorithms for Some Graph Problems

by
K. V. R. C. N. Kishore



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1998

Parallel Algorithms for Some Graph Problems

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
K. V. R. C. N. Kishore

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
April, 1998

CSE-1998-M-KIS-PAR

Entered in System

Certificate

Certified that the work contained in the thesis entitled "*Parallel Algorithms for Some Graph Problems*", by Mr.*K. V. R. C. N. Kishore*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Saxena
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.
March, 1998

Dedicated to
my mother and father

Abstract

In this thesis algorithms are proposed for some graph problems. A new parallel algorithm that runs in $O(\log n)$ time with $O(n)$ cost is proposed for the problem of updating the minimum spanning tree of an undirected underlying graph. A bounded incremental algorithm is also proposed for this problem. Edge insertion and vertex insertion updates are considered.

Constant time parallel algorithms are proposed for testing whether a given graph contains a triangle. Algorithms are also given to recognize (b) claw free graphs and (c) for listing all simplicial vertices in a graph. $O(\log n)$ time algorithms are proposed for testing presence of a incomparability graph on 6 vertices, a forbidden subgraph for partial-3 trees.

Existing optimal algorithm for computing the maximal matching of a general graph in parallel takes $O(\log^3 n)$ time. This also matches the parallel time for computing the maximal matching in a bipartite graph with optimal work. There is also a nonoptimal algorithm which runs in time $O(\log^{2.5} n)$ with $O((m+n)\log^2 n)$ work. We conjecture that computing a maximal matching in bipartite graph should be faster than in a general graph and also prove that the parallel time complexity of maximal matching in general graph is $O(\log\log n t_b(n,m)+t_{so}(n,m))$ where $t_b(n,m)$ is the time for bipartite graph and $t_{so}(n,m)$ is the time for a nonoptimal algorithm for general graphs. Thus, using our result any improvements in the bipartite matching algorithm would reflect in improvement in the parallel time complexity of the problem in general graph.

For edge colouring of graphs we give a constant time algorithm to produce a valid $\Delta^2 \log n$ colouring using the bipartite decomposition of a given graph into edge disjoint bipartite graphs. We also reduce the number of colours to $O(\Delta^2 \log k$ if we

are provided with a vertex colouring using k colours.

For numbers drawn form a restricted universe, called integers, we propos a sub-linear time sequential algorithm to compute the minima. The time taken by the algorithm depends on the word size of the machine and the size of the universe from which the integers are drawn. A parallel algorithm is also obtained for this problem.

Acknowledgments

I express my sincere thanks to Dr. Sanjeev Saxena for his guidance during the course of this work. His valuable advice has helped me in many situations. His patient reading of the report has uncovered many typos that crept.

I also wish to thank my friends, SriRam, Madhu, Kiran and Yugandhar for many discussions with them for the thesis and off the thesis. All of them paid a willing ear many a time during this work. The KLC trio also have given me a good company. Friends of M. Tech96 batch also made my stay during this course a happy one. The lively atmosphere of Hall 4 is hardly forgettable. The evenings at Hall 4 quad remain the happiest ever in my life.

I would also like to extend thanks to my family members for encouraging me and supporting me during the last six years of my stay outside home.

Last but not the least, I would like to thank all those who directly or indirectly contributed towards the successful completion of this work.

For any of the typos or errors I extend my apologies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Parallel processing involves employing more than one processor to perform a given task. There is a strict upper bound on the processor speeds of traditional uniprocessor systems. This bound is dictated by the speed of light and the minimum distance required between two components so that they do not interact with each other. Hence hardware technology alone may not satisfy the ever increasing demands of computational power. Thus parallel computers have emerged as an alternative to overcome this problem.

An algorithm is a sequence of steps directing the computer to solve a particular problem. A sequential algorithm specifies the actions to be taken by a single processor for solving a problem. In a sequential algorithm only a single instruction is executed at any time. In a parallel algorithm, the algorithm is executed by more than one processor simultaneously thus reducing the computation time. But however we have to extract the parallelism inherent in the problem to obtain good parallel algorithms. The next subsection describes in brief the problems chosen for this thesis.

1.1.1 Preliminaries

In this section we discuss the problems considered in this thesis in more detail.

- Updating MST: The problem involves updating the minimum spanning tree

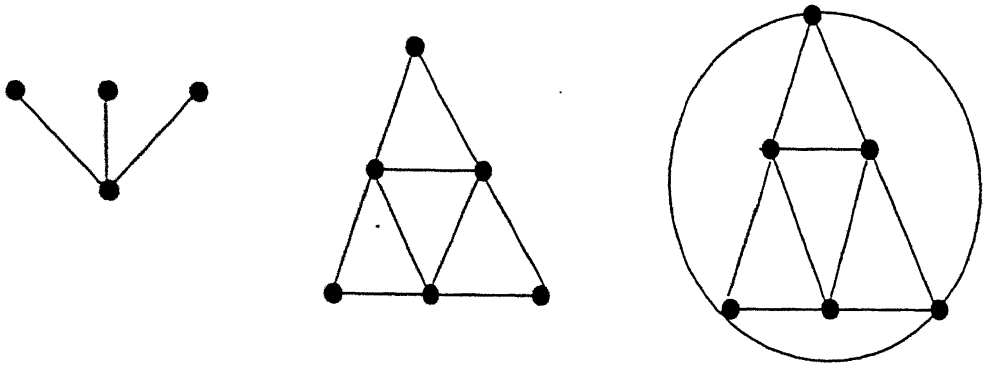


Figure 1: (a) Claw graph, (b) Incomparability graph on 6 vertices and (c) Forbidden subgraph on 6 vertices for partial 3 trees.

of an underlying graph G under edge insertions. That is, given a minimum spanning tree T of the graph G under some weight function $W : E \rightarrow R$ and an edge $e = (u, v)$ with weight $W((u, v))$, find the new minimum spanning tree T' of the graph $G' = G \cup \{(u, v)\}$. We present an $O(\log n)$ time $O(n)$ work EREW algorithm for this problem. We also consider an alternative complexity measure for analyzing dynamic algorithms [44].

- Small Subgraph Recognition: This centers around identifying for the presence of certain subgraphs in a given graph. We consider triangle, claw graph, incomparability graph on 6 vertices. The claw graph, incomparability graph and the forbidden subgraph for partial 3-trees are shown in Figure 1. We also consider the problem of listing all simplicial vertices in a graph. A vertex is called a simplicial vertex if it's neighbourhood is a complete subgraph.

- Maximal Matching: In this problem we are interested in the parallel complexity of maximal matching in a general graph. The study is motivated by the opinion that the case of bipartite graphs is simpler than general graphs. We derive an expression in terms of the time complexity of computing maximal matching in a bipartite graph.

- Edge Colouring: The problem of edge colouring graphs is to assign colours to the edges of a graph G with the restriction that no two edges sharing a common

end point are assigned the same colour. This finds applications in scheduling problems.

- Integer Minima: Here we are interested in finding the minima of a set of integers, drawn from a restricted domain in less than linear time. We also give a parallel algorithm for this problem. The time requirements depend on the word size of the machine and the range of the universe.

1.2 Model of Computation

To analyze algorithms, we need a abstract model of computation on which all the cost and space requirements of the algorithm can be expressed and compared against existing ones for answering questions of efficiency. In this thesis, we use the Parallel Random Access Machine (PRAM) as the model of computation for the parallel algorithms. PRAM is the parallel analogue of the unit cost sequential random access machine. The model involves N processors numbered from 0 to $N-1$. All the processors have access to a memory consisting of cells numbered from 0. Further, the processors are assumed to be aware of their number which is also called its *id*. Depending on the way simultaneous access to a same memory location by more than one processor is allowed, the family of PRAM's is classified into 3 classes as below.

- *E*xclusive *R*ead *E*xclusive *W*rite(EREW): In this model, at any particular instant no more than one processor is allowed to either read or write to the same memory location. The algorithm designer should write algorithms in such a way that such conflicts never occur.

- *C*oncurrent *R*ead *E*xclusive *W*rite(CREW): In this model simultaneous read of a memory location by more than one processor is allowed but simultaneous write to the same memory location in forbidden.

- *C*oncurrent *R*ead *C*oncurrent *W*rite(CRCW): This model allows both simultaneous read as well as write of same memory locations by more than one processor. According to the resolution method of Concurrent write this is further subclassified into the following models.

- COMMON: All processors writing to a same location should write the same value.
- PRIORITY: The smallest numbered processor succeeds in the write.
- ARBITRARY: One processor is gaurenteed to succeed but no commitment is made as to which processor succeeds.

A parallel algorithm that runs in time $O(t(n))$ using $p(n)$ processors is said to have a cost of $c(n) = O(t(n)p(n))$. If this cost equals the cost of the best known sequential algorithm,then the parallel algorithm is said to be work optimal [30].

In chapter 6, we however deviate from this model of computation and use the nonconservative RAM model. In this model, the word size supported by the machine is bigger than in the RAM. Specifically, the word size of the machine is assumed to be logarithmic (upto a constant) in input. Thus for inputs of size of N, RAM is supposed to have a word size of $O(\log N + \log n)$ where n is the size of the problem. The nonconservative RAM is assumed to have a word size of $O(\log N \log^c n)$ for some constant c. This model too has a parallel analogue named the nonconservative PRAM and all the concepts and classifications of PRAM apply to this model also.

1.3 Overview of the thesis

The rest of the thesis is organized as follows. In Chapter 2, we discuss the algorithms for updating the minimum spanning tree of an underlying graph subject to edge insertions. In Chapter 3, we present the algorithms for recognition of small subgraphs. Algorithms are presented for recognition of triangle, claw free graphs, an incomparability graph on 6 vertices, a forbidden subgraph for partial 3−trees and also for listing simplicial vertices in a graph.(See Figure 1)

In Chapter 4 we consider the problem of computing a maximal matching in a general graph. We obtain an expression stating that this problem requires $O(\log\log n t_b(n,m) + t_{so}(n,m))$ time where $t_b(n,m)$ is the time for bipartite matching algorithm and $t_{so}(n,m)$ is the time taken by the existing nonoptimal algorithm for general graphs. Since our result is optimal, any new optimal algorithm for bipartite graphs results in a better optimal algorithm for general graphs also.

In Chapter 5, we present the algorithms for edge colouring of graphs. Our algorithm runs in constant parallel time and produces a $\Delta^2 \log n)$ colouring. We show an explicit construction for this.

Chapter 6 differs significantly form the above chapters in that the model of computation we use is the nonconservative RAM (and nonconservative PRAM) where the word size of the machine is assumed to be substantially bigger. We make use of this inherent power in the machine to obtain a sublinear sequential algorithm for finding the minima of n integers. We also present a parallel algorithm for this problem on the parallel analogue of this nonconservative RAM which is called the nonconservative PRAM.

In Chapter 7 we offer some concluding remarks and state some of the possible extensions from our work.

Chapter 2

Parallel and Bounded Incremental Minimum Spanning Tree

2.1 Introduction

Graphs play a key role in many problems related to computer science because of their numerous applications. Dynamic graph problems too have received great attention [17]. Dynamic problems involve updating the solution subject to changes in input without building from scratch. In this chapter we consider the problem of updating the minimum spanning tree of a graph after edge insertions. More formally, we define the problem as follows. Given a graph $G = (V, E)$ with a weight function $W : E \rightarrow R$ and a minimum spanning tree(MST) T of G, update the MST when an edge $e = (u, v)$ with weight $W(e)$ is added to G. According to terminology of Eppstein *et. al.* [17] the problem can be called partially dynamic incremental MST update.

The problem of updating an MST has good algorithms in the sequential model of computation. Fredrickson [21, 20] use a data structure called *topology trees* which supports update operations in $O(\sqrt{m})$ time. Some special cases when the underlying graph is planar are also considered by Eppstein *et al.*[18].

The problem of MST update on a PRAM has been extensively studied. Pawagi and Ramakrishnan [43] consider MST update subject to two modifications namely increase and decrease in edge weight and adding a vertex z along with edges incident

from z. They present $O(\log n)$ time algorithm on CREW PRAM with $O(n^2)$ processors for both these problems. The same algorithm also supports edge insertions as well as deletions within the same bounds. Although their algorithms are very much uniform in nature for all the problems, they are very costly because of the amount of information that they compute in the form of tables (F^+ and M^+) for every update. Johnson and Metaxas [32] considered the single and multiple vertex update problem. They present $O(\log n)$ time $O(n)$ cost algorithms for the single vertex update problem on the EREW PRAM. Here we present an $O(\log n)$ time $O(n)$ cost EREW algorithm for the single edge insertion MST update problem. We thus improve the algorithm of Pawagi and Ramakrishnan [43].

In the second part of the Chapter, we treat the same problem and present a bounded incremental algorithm for updating the minimum spanning tree of an underlying graph under edge insertions. The complexity of the algorithm is measured in terms of the amount of change that the current solution has to undergo. This parameter is denoted by $\|\delta\|$ and the requirements of the algorithm is analyzed in terms of size of $\|\delta\|$ denoted by $|\delta|$. This complexity measure was introduced by Ramalingam [44] for analyzing dynamic algorithms.

The rest of the Chapter is organized as follows. The EREW algorithm with the same time and cost bounds is presented in Section 2.2. In Section 2.3, a bounded incremental algorithm for this problem is presented. This algorithm does work linearly proportional to the amount of change.a In Section 2.4 we consider the case of single vertex update.

2.2 EREW MST Update

In this section we present an EREW algorithm to update minimum spanning tree in $O(\log n)$ time and $O(n)$ cost. The algorithm uses routines for numbering the vertices of a tree in preorder and postorder [46] and LCA computation of a pair of vertices [45]. These operations can be done in $O(\log n)$ optimal EREW time. Without loss of generality, we assume that for the edge (u, v), u occurs before v in the preorder traversal of the tree.

The key idea in this algorithm is to locate the edges which are part of the cycle formed after the new edge has been added. Preorder and postorder numbering of the vertices of the tree is used to identify the unique path between the vertices u and v in the original minimum spanning tree T. Our discussion is based on the Figure 3.

If u is an ancestor of v, the nodes in the unique path between the given pair of vertices u and v are the vertices whose preorder number as well as the postorder number lie in between the preorder and postorder number of the vertices u and v. (See Figure 3 and Theorem 2.1.)

Theorem 2.1 *In a rooted tree T, for any two vertices u and v such that u is an ancestor of v, a node is in the unique path between the vertices u and v if its preorder number lies in the range [preorder(u),preorder(v)] and its postorder number lies in the range [postorder(v),postorder(u)].*

Proof: We first prove the following result. In a rooted tree T, a node r is an ancestor of node u if the preorder number of r is less than the preorder number of u and the postorder number of r is greater than the postorder number of u i.e. preorder(u) < preorder(r) and postorder(r) > postorder(u). To prove the above statement, we first define "left", "right", "above" and "below" order on the nodes consistent with the embedding of the tree T. By the definition of the preorder number, any node r which has preorder number less than the preorder number of node u must be either "above" u or a "left" sibling of u or a node in the subtree rooted at one of the "left" siblings of u. Similarly, by the definition of postorder numbering, a node r with postorder number greater than the postorder number of u is either a node "above" u, or a "right" sibling of u or a node in the subtree rooted at the "right" siblings of u. Thus, the set of nodes satisfying both the conditions is the set of ancestor nodes. (See Fig 2.)

We continue with proof of the theorem. Consider nodes u and v. The set of ancestors of v are a subset of ancestors of node u. Moreover, the set of ancestor nodes of v have preorder number less than that of v and postorder number greater than that of v (from the above result). The same holds for ancestors of u also. As u is an ancestor of v, $preorder(u) < preorder(v)$ and $postorder(u) > postorder(v)$. Thus the nodes in the path from u to v have preorder number which is greater than that

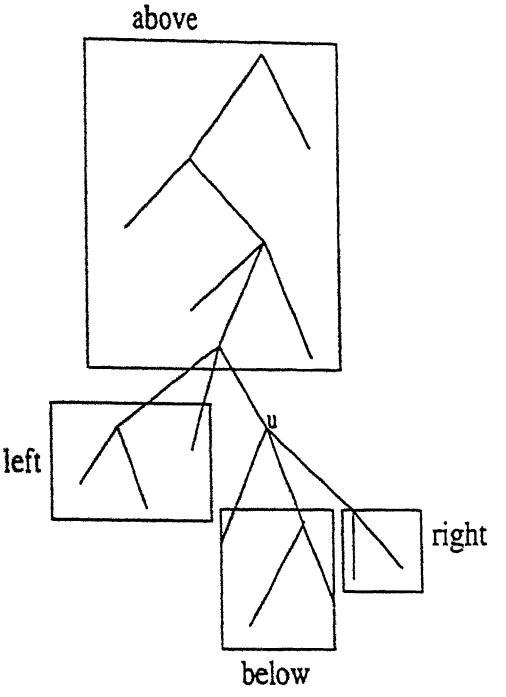


Figure 2: The "left", "right", "above" and "below" regions of node u.

of v and less than that of u. Similarly, the postorder numbers of the nodes between u and v lies in the range [postorder(u),postorder(v)]. Thus if we consider the nodes with the numbers in the range specified by the theorem, we will be left with the set of nodes between u and v. ∎

Theorem 2.1 gives an efficient way to compute the path between a node and its ancestor in the tree T. If it is the case that neither is an ancestor of the other, then observe that the path from u to v is the concatenation of the path from u to $LCA(u, v)$ and $LCA(u, v)$ to v.

An example is shown in Figure 3. Here u and v are neither an ancestor nor a descendant of the other. So we compute $LCA(u, v)$, which is x, and compute the two paths from x to u and from x to v. The path is shown in heavy lines.

These observations lead to the following algorithm.

procedure EREW_Update
comment Given a graph $G = (V, E)$, a rooted minimum spanning tree T of G and an edge $e = (u, v)$ being added to G.

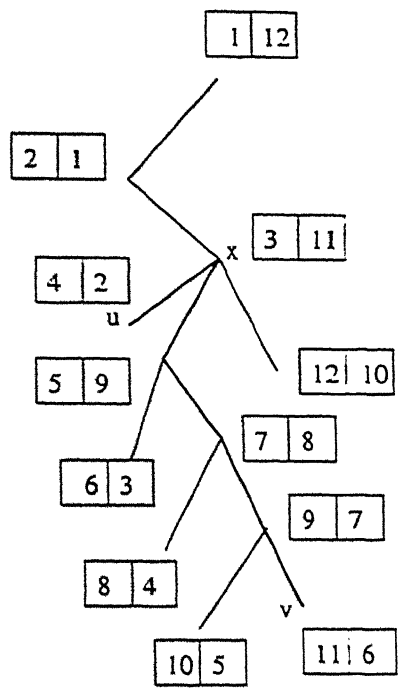


Figure 3: Identification of unique path between a pair of vertices u and v in a tree T. The number to the left in the box associated with each node gives it preorder number and the number to the right is its postorder number

begin

Step 1. for every node of the tree T obtain the *preorder()* and *postorder()* numbers of the nodes and the parent $P()$ of each node.

Step 2.0. let $x = LCA(u, v)$

Step 2.1for every vertex r do in parallel

if (preorder(r) $\in$ (preorder(x),preorder(v)] and postorder(r) $\in$ [postorder(v),postorder(x)))

In_Path(r) = True;

endfor;

Step 2.2. if $LCA(u, v) \neq u$ then compute the path between $LCA(u, v)$ and u as in Step 2.1.

comment Identify the edge to be removed. Note that if vertex r is in the path, then the corresponding tree edge is $(r, P(r))$.

comment Array M of size n with each entry initialized to $-\infty$ is used.

Step 3.for all vertices r do in parallel

if In_Path(r) then

$M[Preorder(r)] := Wt(r, P(r))$;

Step 4.Find the maximum value in the array M and remove the corresponding edge from the tree.

end.

A simple analysis of the algorithm shows that Steps 2,3, and 4 can be done in $O(1)$ time with $O(n)$ processors and Step 1 in $O(\log n)$ time and $O(n/\log n)$ processors. So the whole algorithm takes time of $O(\log n)$ with $O(n/\log n)$ EREW processors and the following Theorem follows.

Theorem 2.2 *The minimum spanning tree of a graph $G = (V, E)$ can be updated in $O(n/p + \log n)$ time with p processors on a EREW-PRAM model when an edge $e = (u, v)$ with weight $W(e)$ is added to G.* ∎

2.3 Bounded Incremental Algorithm

In this section, we present a sequential algorithm to update the minimum spanning tree of a underlying graph G subject to edge insertions. The time complexity of the algorithm is measured in terms of the amount of change that the current solution has to undergo in order to correctly reflect the new state. Ramalingam [44] introduces this notion of complexity measure for incremental algorithms and gives incremental algorithms for several graph problems where complexity is measured in terms of the parameter namely the amount of change denoted by $||\delta||$. In the subsection 2.3.1 we briefly discuss this measure of complexity and the algorithm is given in subsection 2.3.2.

2.3.1 Bounded Incremental Computation

In general, an incremental graph problem $\mathcal{P}$ takes as input G, the solution to graph G and input change δ. The job of the algorithm is to compute the solution for the graph $G \cup \{\delta\}$ where $G \cup \{\delta\}$ is the graph obtained by making a change δ to G. Here we measure the time and space requirements of the algorithm not in terms of the conventional *size of input* but as a measure of the amount of change in input and output required to modify the solution.

More formally, a vertex $u \in G$ or $G \cup \{\delta\}$ is said to be a modified vertex if δ inserted u or deleted u. (In our case, only insertions need be considered.) The set of modified vertices is denoted by the set $\text{MODIFIED}_{G,\delta}$. δ creates a set of vertices that are affected by the change. With reference to the problem under consideration a vertex is said to be affected iff that node is in path(u, v) in the given minimum spanning tree T. We call this set as $\text{AFFECTED}_{G,\delta}$. Similar definitions for the set of edges modified and affected can be given. Let $\text{CHANGED}_{G,\delta} = \text{MODIFIED}_{G,\delta} \cup \text{AFFECTED}_{G,\delta}$. This set captures the change in input and output. When there is no confusion, we can do away with the subscripts. Finally, $||\delta||$ is the union of the sets capturing CHANGED_δ of both $V(G)$ and $E(G)$. This parameter $||\delta||$ captures the amount of change the current solution has to undergo for accounting to change in the underlying graph G. The complexity of the incremental algorithm is measured

in terms of the size of this parameter namely $|\delta|$.

An incremental algorithm is said to be bounded if the time taken by the incremental algorithm to process a change δ to the problem $\mathcal{P}$ is bounded by some function $F(\|\delta\|, \mathcal{P})$. Otherwise it is said to be unbounded. A dynamic problem is said to be (un)bounded if it has(*resp.* doesn't have) a bounded incremental algorithm, that is there is an(no) incremental algorithm that runs in time which is a function of δ independent of the input size.

For a measure of efficiency among incremental algorithms, classification may be based on the time requirement in terms of the parameter $|\delta|$. That is, an incremental algorithm is said to be polynomially bounded if its time complexity can be expressed as a polynomial of δ. Otherwise, it is said to be exponential.As can be inferred, incremental algorithms which have complexity of low-order polynomial of δ are "good". In the next section we apply these ideas to obtain an $O(|\delta|)$ incremental algorithm for the update of minimum spanning tree.

2.3.2 Bounded Incremental Algorithm for MST Update

In this subsection we assume that the input is in the form of (node,parent) pairs. The assumption that the input tree T is rooted is vital as will be explained later. By a rooted tree we mean that each node knows its parent. Moreover, after the algorithm ends, the output that we generate should also be a rooted tree.

If an edge $e = (u, v)$ is added to G whose MST T is to be updated, we can define MODIFIED$_\delta = \{u, v\}$. Similarly, the addition of this edge (u, v) may result in change of parent pointers of nodes in the u, v path in the tree. Thus, AFFECTED$_\delta$ = $\{r | r \in T$ and r lies in the unique (u, v) path $\}$ where T is the input to the problem in the form of a rooted tree. The set of affected vertices and edges are shown in Figure 4 after insertion of the edge (u, v).

Applying the notion of complexity measure in terms of amount of change δ to our problem, we note that the value of δ is given by the length of the path between the vertices u and v in the current minimum spanning tree T. The length of the path is given by Preorder(u) + Preorder(v) - 2* Preorder($LCA(u, v)$) - 2.

Computing the path using Theorem 2.1 takes $O(\delta)$ time for this part. But the

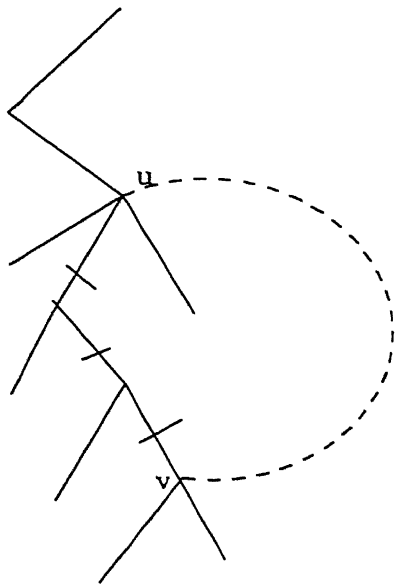


Figure 4: The minimum spanning tree T along with the edge added shown in dotted lines. The set of AFFECTED edges are shown by a cross across them. The AFFECTED vertices are shown with a '*' over them.

computation of preorder and postorder numberings cannot be generated within the same bounds. We instead avoid those explicit computations. We depend on the LCA computation of two nodes u and v in a rooted tree T. That is, traverse the path from v and u along their parents using knowledge of $P()$, every time checking for the LCA(u, v) node. This process takes time which is $O(\delta)$, 2δ to be precise. During this process, we need to keep track of whether a node is seen by u during traversal or not. Thus during each step, the node visited by u is flagged and v does a checking operation for the node it visits most recently. So, it is assumed that each node has another unit space for storing this flag. As we keep track of the path traversed by u during this path computation, we can at the end of the algorithm reset these flags. Effectively, we do not use any extra space that is to be charged to the algorithm. We use only $O(\delta)$ space for this whole process. To obtain the complete u, v path, we need to concatenate these two paths. (see Aho *et al.*[1] for data structures to support such operations). This path is stored for later use. It is during this computation that the knowledge of *parent* is required. Otherwise, we cannot do this computation in the stated time bounds.

In the obtained path after we find the heaviest edge and compare its weight with $W(e)$. One of them remains in the minimum spanning tree and the other leaves the tree. We now address the restructuring of the tree T after this update. There are two cases. The trivial case is when T undergoes no change and clearly nothing needs to be done. In the other case, let the tree edge (p, q) leave the tree. Let (p, q) be in the path from $LCA(u, v)$ to v. We assume that $p = P(q)$. Now we update the parent pointer of all the nodes in the path q to v as follows. Label the nodes in the path from q to v as $q = q_0, q_1, \cdots, q_k = v$. Earlier, $P(q_k) = q_{k-1}$ and $P(q_i) = q_{i-1}$ for $1 \leq i \geq k-1$. Now, we set $P(v = q_k) = u$ and $P(q_{i-1}) = q_i$ for $2 \leq i \leq k-1$. (See Figure 5.) In the other case, if an edge in the path from $LCA(u, v)$ to u is the edge leaving the tree, then the following restructuring has to be done. Let the edge (p, q) leave the tree where $p = P(q)$. Label the path from q to u by $q = q_0, q_1, q_2, \cdots, q_k = u$. Set $P(q_i) = q_{i+1}, 0 \leq i < k$ and $P(q_k) = v$.

The above discussion can be captured in the following algorithm.

```
Algorithm Update_MST(G,T,e);
Input: The underlying graph G. the current minimum spanning tree
T and the edge e = (u, v) being added to G.
Output: The new minimum spanning tree.
initialize cur_u = u and cur_v = v;
mark(cur_u) = 1; mark(cur_v) = 1;
index = 0;
comment The while loop computes the unique u-v path in T.
while not done
      cur_u = P(u);
      mark(cur_u) = index;
      index = index + 1;
      add cur_u to u_path;
      cur_v = P(v);
      add cur_v to v_path:
      if mark(cur_v) = 1 then
        concatenate the paths u_path(1:mark_u(cur_v)) and v_path to obtain the
```

u-v path in the tree T and store the vertices in the path in path_uv;
done = true;
end {while}
find the maximum cost edge among the edges $(r, P(r)$ for $r \in$ path_uv
and remove that edge;
comment: restructure the tree
if the edge (p, q), $p = P(q)$, in the path $LCA(u, v)$ to v is the leaving edge
Label the nodes in the path from q to v as $q = q_0, q_1, \cdots, q_k = v$;
set $P(v = q_k) = u$;
for $2 \leq i \leq k - 1$ do $P(q_{i-1}) = q_i$;
if the edge (p, q), $p = P(q)$, in the path $LCA(u, v)$ to u is the leaving edge
Label the path from q to u by $q = q_0, q_1, q_2, \cdots, q_k = u$;
$P(q_k) = v$;
for $0 \leq i < k$ do $P(q_i) = q_{i+1}$;
end. {Algorithm}

From the above discussion, the following theorem follows.

Theorem 2.3 *Partially Dynamic Incremental MST Update of a general underlying graph is incrementally bounded. In particular, there is an $O(\delta)$ time algorithm for performing the update of a MST under edge insertions using only $O(\delta)$ space.* ∎

Thus after the algorithm ends, we have restructured the input tree to be a rooted tree. To be more precise, we start with a tree in which every node knows its parent and after the update, the same holds true. This restructuring is also important as rooting a tree takes time proportional to the number of nodes in the tree which our algorithm cannot afford to. Moreover, we make use of extra space associated with each node during the algorithm and at the end of the algorithm clear the space. If this clearing is not done, the space requirement grows to $O(n)$ or the time requirement grows to $O(\delta \log \delta)$. If there is a data structure that supports Insert(x) and LookUp(x) in constant time where the values that are stored in the data structure are in some suitably restricted domain. then we need only $O(\delta)$ space and time. There are data structures that support constant time lookup on static tables. (see [47]).

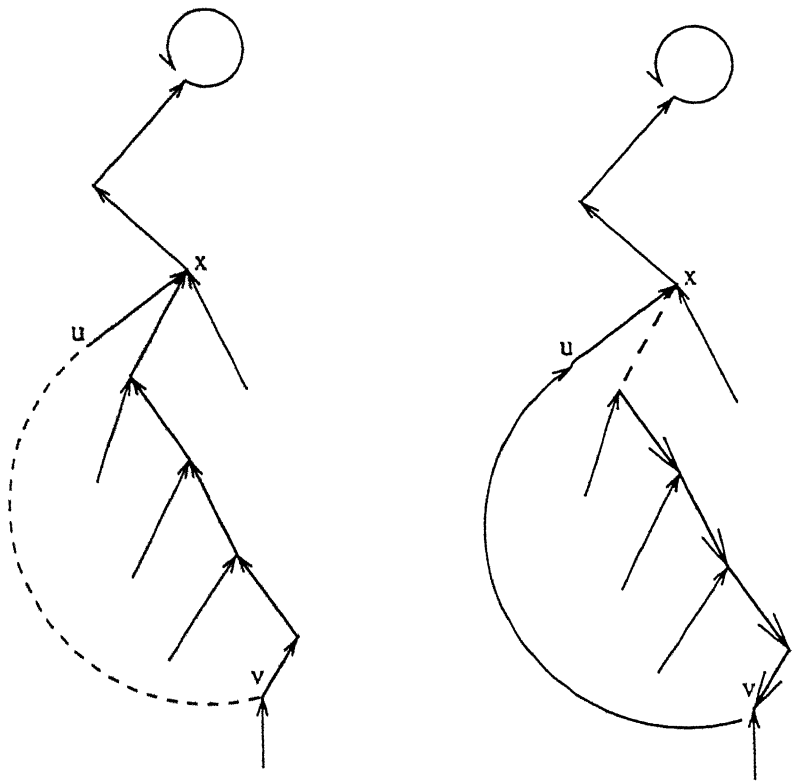


Figure 5: Updating the parent pointers along the path from u to v. The dashed edge is the edge that leaves T and (u, v) enters the tree.

2.4 Single Vertex Update Problem

In this Section, we give a bounded incremental algorithm for updating the MST of an underlying graph after a vertex z along with edges from z are added to the graph G. The edges from z are associated with weights from the weight function W. The algorithm is an extension of the algorithm presented in Section 2.3 to the case of single vertex update.

We have to first find the minimum weight edge among the r edges incident from z and make it a tree edge. This does not violate the minimality of the MST of the new graph because of the property of the MST that from every vertex the edge of minimum weight will always be in the MST. Then we proceed to break the cycles in the new tree $T \cup \{(z, z_i)|W(z, z_i) = min_j\{W(z, z_j)\}, j = 1, 2, \cdots, r$. We describe the algorithm in detail below.

Algorithm Single-Vertex-Update(G, T, z)
Input: The underlying graph G, the current spanning tree T and the vertex z along with r edges $z_1, z_2, \cdots, z_r$ from z.
Output: The MST for $G = (V \cup \{z\}, E \cup \{(z, v)|v = z_i, i = 1, 2, \cdots r\})$
Step 1. Add the minimum weight edge of z to T as tree edge
Step 2. for $i = 1$ to r do
Identify the cycle formed by the edge (z, z_i) in T
comment This can be done in time proportional to the length of the cycle induced by (z, z_i) using the algorithm in previous section
Step 3. Remove the heaviest edge in each cycle
Step 4. Repeat Steps 2 and 3 until all cycles are removed
comment Step 3 is required because if cycles thus formed in Step 1 intersect then the breaking of each cycle fails to remove all the cycles. (See Figure 6)
end.{Algorithm}

We now analyze the time required by the above algorithm. The time for Step 1 is $O(r)$. For Step 2, using the algorithm in the previous section, the time required is $\sum_{i=1}^{r-1} \delta_i$ where δ_i is the length of the cycle of (z, z_i) . Step 3 also takes the same time.

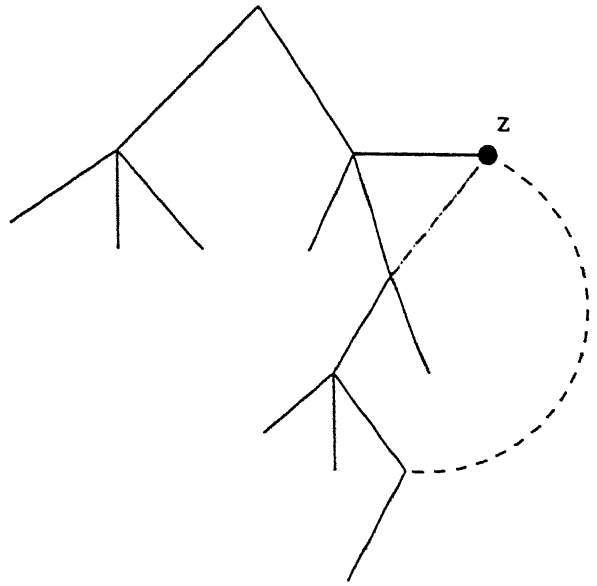


Figure 6: Showing the formation of cycles during single vertex update. The bold edge coming from z is the tree edge at z and the dashed edges are the edges that induce cycles in T. The weights of the edges are not shown in the figure.

But these steps have to be repeated $r-1$ times before all cycles are broken. Thus the time for the entire algorithm is $O(r\sum_{i=1}^{r-1}\delta_i)$.

2.5 Conclusions

We have described an $O(n)$ work $O(\log n)$ time parallel algorithms for incrementally constructing the spanning tree of a general underlying graph. Our algorithm can be easily seen to support the query of the type whether the edge (u,v) is a tree edge or not in constant time and also query about the the weight of the current minimum spanning tree is supported in constant time. However, the algorithms do not support edge deletion.

Chapter 3

Efficient Parallel Recognition of Small Induced Subgraphs

In this Chapter we give parallel algorithms for finding small subgraphs in parallel efficiently. We first give constant time parallel algorithms for finding whether a given graph contains a triangle as a subgraph or not. This is used to recognize claw free graphs in $O(\log n)$ parallel time with a cost of $O(m^{1.5})$. A sequential simulation of our algorithm improves the cost of the existing sequential algorithm from $O(m^{1.69})$ to $O(m^{1.5})$. Two algorithms for listing all simplicial vertices in a graph are also given. Algorithms to recognize two subgraphs on six vertices are also presented.

3.1 Introduction

An interesting problem is to check whether a given graph contains a triangle or not. Itai and Rodeh [29] present two algorithms for this problem. The running time is $O(n^{\alpha})$ and $O(m^{3/2})$, where α is the exponent of matrix multiplication. Later an $O(m^{\frac{2\alpha}{\alpha+1}})$ algorithm for this problem is given by Alon et. al. [3] Here we give a new analysis to this problem to show that this problem can be solved in time $O(n\sigma^2 + m^2/n)$ sequential time where σ^2 is the variance of the degree sequence of the given graph. For sparse graphs, $m < n^{1.6}$ and for graphs where the variance of the degree sequence is small our result is an improvement of the existing result.

We also present $O(1)$ time parallel algorithms for this problem using $O(n\sigma^2 + m^2/n)$ processors.

We give parallel algorithms for recognizing claw free graphs. The algorithm runs in $O(\log n)$ time. The cost of the algorithm is $O(m^{1.5})$ which improves the sequential algorithm of Kloks *et al.* [37]. It is shown that matrix multiplication can be used in algorithms for recognition of graphs with two subgraphs on six vertices are also given. An $O(\log n)$ time algorithm for listing all simplicial vertices of the graph is also given. An alternative algorithm for this problem that runs in constant time with a small deterioration in cost is also given.

We note that we eliminate the bottleneck of matrix multiplication in most of our algorithms. This gains further importance because of the constant parallel time achieved.

The rest of the Chapter is organized as follows. In Section 3.2 we present the algorithm for recognizing the existence of a triangle in a graph. This is used in Subsection 3.2.1 to recognize claw free graphs. In Section 3.3 the existence of triangles is used to identify the subgraphs on six vertices. Finally, Section 3.4 gives the algorithms for listing all simplicial vertices of a graph.

3.2 Recognition of Triangle

Itai and Rodeh [29] gave the following algorithm for checking whether a given graph contains a triangle. Let M be the adjacency matrix of the graph. Compute M^2 the square of the adjacency matrix. $M^2(u,v) = 1$ iff there exists a vertex w such that $M(u,w) = 1$ and $M(w,v) = 1$. If $M(u,v) = 1$ then there exists a triangle passing through the edge (u,v) of the graph. Let $B = M^2$ AND M, the boolean bit wise AND of corresponding entries of M^2 and M. The (u,v) entry of B contains a 1 iff there is a triangle through (u,v).

A parallel algorithm for this can be made to run in $O(\log n)$ time using $O(n^\alpha)$ processors. This is because matrix multiplication can be done in $O(\log n)$ time on CREW PRAM using $O(n^\alpha)$ processors [13]. We give an alternative algorithm that runs in constant parallel time.

The algorithm is to check at each vertex whether there is a triangle containing this vertex. This can be done in time $O(d(v)^2)$ where $d(v)$ is the degree of the vertex v. So in the sequential case, the time is $O(\sum_{v \in V} d(v)^2)$. We now bound this expression as follows.

From statistics,

$$\sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \overline{X})^2}{n}$$

where σ is the standard deviation and $\overline{X}$ is the mean of the sample $x_1, x_2, \cdots, x_n$ of size n. If we let each x_i be the degree of vertex i of the graph then $\overline{X} = \frac{\sum_{i \in V} x_i}{n} = \frac{2m}{n}$. Thus,

$$\begin{aligned} n\sigma^2 &= \sum_{v \in V} d(v)^2 - 2d(v)\overline{X} + (\overline{X})^2 \\ &= \sum_{v \in V} d(v)^2 - \frac{4m^2}{n} \end{aligned}$$

Hence we can bound the sequential time by

$$\sum_{v \in V} d(v)^2 = n\sigma^2 + \frac{4m^2}{n}$$

Observe that $\sum_{v \in V} d(v)^2 \leq \Delta \sum_{v \in V} d(v) = m\Delta_{max}$ where Δ_{max} is the maximum degree of any vertex of G.

The best algorithm Alon [3] takes time $O(m^{\frac{2\alpha}{\alpha+1}})$. The cost of our algorithm is $O(n\sigma^2 + \frac{4m^2}{n})$. Our algorithm has less cost if $m < n^{\alpha+1}2$ and $\sigma^2 < n^{\alpha-1}$. Currently, $\alpha \approx 2.4$. Thus our algorithm is better if $m < n^{1.6}$ and $\sigma^2 < n^{1.4}$. Moreover For random graphs, Bollabas [6] has shown that the variance of the degree sequence equals the mean of the degree sequence. Thus, $\sigma^2 = \frac{2m}{n}$ and hence for random graphs the cost of our algorithm would be $O(\frac{m^2}{n})$. But form the parallel algorithmic point of view our algorithm runs in constant time whereas a direct parallelization of the algorithm of Alon *et al.* [3] cannot run in time less than $O(\log n)$. This is because of the bottleneck of matrix multiplication [13]. Thus our algorithm is *fully parallelizable* even at some modest increase in cost in some extreme cases. The cost of our algorithm exceeds the cost of the algorithm of [3] only when $m > n^{1.7}$ with the value of $\alpha \approx 2.4$. As will

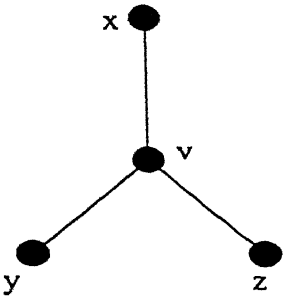


Figure 7: Claw graph. The vertex v of degree 3 is called the central vertex.

be shown in the next subsection, our constant time algorithm is useful in some cases and results in a constant time parallel algorithm for recognition of claw free graphs.

A constant time parallel algorithm for this problem is described below. At every vertex we allocate $d(v)^2$ processors. The computation at every vertex v is to check for all pairs of neighbours of v, say u, w, whether $(u, w) \in E$. Thus we can detect the presence of a triangle in a graph in constant parallel time. The graph can be represented in the form of adjacency lists. The following theorem follows from the preceding discussion.

Theorem 3.1 *Given a graph $G = (V, E)$, it can be checked in $O(1)$ parallel time whether G contains a triangle or not.* ∎

3.2.1 Recognition of Claw Free Graphs

In this section we give a parallel algorithm for recognizing claw free graphs. The claw is the graph shown in Figure 7. The importance of the claw graph comes from matching properties, line graphs, comparability graphs etc. Any graph containing claw as an induced subgraph cannot be transitively oriented. Thus claw is a subgraph of incomparability graphs [24]. A graph is said to be claw free if it does not have an induced subgraph isomorphic to a claw (see Figure 7). We call the vertex of degree 3 in a claw as the central vertex. The following theorem characterizes the claw free graphs [7, 37].

Theorem 3.2 *If G is a claw free graph then every vertex can have degree atmost $2\sqrt{e}$.*

Proof: Consider a vertex v. If G is claw free then $\overline{G_{N[v]}}$ is triangle free. Let $p = |N[x]|$. From Turán's Theorem, [7], a graph with p vertices and without triangles can have atmost $\frac{p^2}{4}$ edges. Hence there must be atleast $\frac{1}{2}p(p-1) - \frac{1}{4}p^2 = \frac{1}{4}p^2 - \frac{1}{2}p$ edges in $G_{N(v)}$. Then $G_{N[v]}$ must contain atleast $\frac{1}{4}p^2 - \frac{1}{2}p + p \geq \frac{1}{4}p^2$ edges. This can be atmost the number of edges of G. Hence $p \leq 2\sqrt{m}$. ∎

Thus the recognition algorithm proceeds as follows. Compute the degree of every vertex v. If there are vertices of degree greater than $2\sqrt{e}$ then the graph is not claw free. In this case the graph contains a claw with some vertex of degree 3 as the central vertex. Otherwise for every vertex consider the graph induced by complement of its neighborhood and check for the presence of a triangle in the resulting graph. If no such graph contains a triangle then the original graph is claw free.

The time for constructing the complement of the neighborhood is constant. The cost is dominated by verification of triangle in the complement of neighborhood of every vertex v. Let $C(m,n)$ be the cost of the entire algorithm.
Then,

$$\begin{aligned} C(m,n) &= \sum_{v \in V} d(v)^2 \\ &\leq \sum_{v \in V} d(v)\sqrt{m} \\ &\leq m\sqrt{m} \end{aligned}$$

In the above equations the first inequality follows because every vertex has atmost $2\sqrt{m}$ neighbours and we are using the constant time triangle recognition algorithm of Section 3.2. The rest of the equalities follow easily.

Thus the algorithm runs in $O(\log n)$ parallel time. The algorithm we presented in this section improves the sequential algorithm of [37]. The algorithm of Klok's *et al.* [37] takes time $O(m^{\frac{\alpha+1}{2}})$ whereas our algorithm has only $O(m^{1.5})$ cost. In parallel setting our algorithm requires $O(\log n)$ time. We are able to eliminate the bottleneck of matrix multiplication in the case of parallel algorithms. The cost of the algorithm matches the cost of the sequential algorithm.

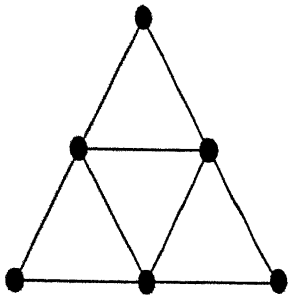


Figure 8: The graph IC_6. It is a incomparability graph on six vertices.

3.3 Subgraphs on Six Vertices

In this section we give recognition algorithms for two subgraphs on six vertices. This gives algorithms for recognizing graphs that are free of the forbidden subgraphs also which we are considering. The algorithms are based on matrix multiplication.

3.3.1 Recognition of IC_6

The graph IC_6 (See figure 8 is another forbidden subgraph of comparability graphs [24].

We give the following recognition algorithm for IC_6.

Algorithm Recognize-IC_6()
begin
Step 1: Let A be the adjacency matrix of the graph
Step 2: Compute the square of the adjacency matrix into B
Step 3: Compute $T = B \wedge A$.
comment From [29] $T(u, v) = 1$ implies there is a triangle through (u, v)
Step 4: For each triangle in the graph do
for each edge (u, v) of the triangle do
compute the list $L(u, v) = \{w | u, v$ and w form a triangle in $G\}$
Step 5: From the lists computed above check for IC_6 at every triangle (uvw) by checking whether there is another triangle through (u, v), (u, w) and (v, w)
end.{Algorithm}

The time taken by the above algorithm is $O(n^\alpha + n^2\Delta)$ where Δ is the maximum

degree of G. The term n^a is for Step 1 and the term $n^2\Delta$ is the time requirement of Step 4. In the case of parallel algorithm, the time required will be $O(\log n)$ and the cost is $O(n^a + n^2\Delta)$.

3.3.2 Recognition of M_6

A $k-tree$ is a graph that can be reduced to the $k-$ complete subgraph K_k by a sequence of operations of "pruning a leaf", that is, removal of a degree k vertex with completely connected neighbours. A partial $k-tree$ is any subgraph of a $k-tree$ or, equivalently, a graph embeddable in a $k-tree$ with the same vertex set [5]. The motivation for studying partial $k-trees$ comes from many applications such as reliability of communication networks in the presence of constrained line and site failures, concurrent broadcasting in a common medium network and other applications. It is shown that many problems that are $NP-$Complete for general graphs have linear time algorithms for partial $k-trees$ when k is fixed.

A graph H is said to be a minor of graph G iff it can be obtained from G by a finite sequence of edge-extraction and edge-contraction operations. Given a graph G and an edge e of G, *edge extraction* results in the graph $G-e$ with the same vertex set as G and the edge set $E(G)-\{e\}$. *Edge Contraction* results in the graph G/e with the same vertex set obtained from the vertex set of G by replacing the end points of e by a new "composite" vertex. This composite vertex inherits all the neighbours of the two replaced vertices without introducing a self-loop or multiple edges.

It was conjectured by Wagner (see [33]) that for every infinite set of graphs, one of its members is a minor of the other. Thus, every class of graphs that is closed under minor taking has a complement with a finite set of forbidden minors. It was argued in [5] that the graph M_6 is a forbidden subgraph of 6 vertices for the class of partial 3 trees [5]. The graph M_6 is shown in Figure 9. We give a recognition algorithm for the graph M_6. The algorithm is very similar to the algorithm for recognition of IC_6.

Algorithm Recognize-M_6()
begin
Step 1: Let A be the adjacency matrix of the graph
Step 2: Compute the square of the adjacency matrix into B

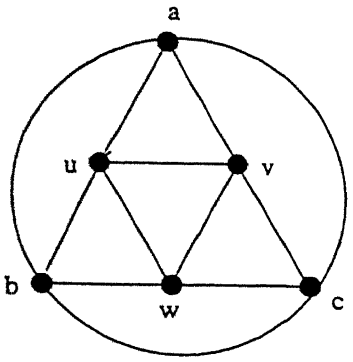


Figure 9: The graph M_6. This is a forbidden subgraph for partial-3 trees on six vertices.

Step 3: Compute $T = B \wedge A$.

comment From [29] $T(u, v) = 1$ implies there is a triangle through (u, v)

Step 4: For each triangle in the graph do

for each edge (u, v) of the triangle do

compute the list $L(u, v) = \{w | u, v$ and w form a triangle in $G\}$

Step 5: From the lists computed above check for M_6 at every triangle

comment This is done as follows. At every triangle (uvw) check whether there are triangles from (u, v), (u, w) and (v, w) other than this triangle. From all such triangles check again whether there is a triangle connecting the vertices a, b and c (See Figure 9)

end.{Algorithm}

The time taken will be $O(\log n)$ with cost $O(n^\alpha + n^2\Delta)$. This can be proved as in subsection 3.3.1. The algorithms in this section are greatly simplified by knowing the presence of triangle through a given edge (u, v) of the graph G.

3.4 Listing All Simplicial Vertices

For many graph problems simplicial vertices can be safely removed, tackling the problem on the reduced graph. Thus it is desirable that we identify all the simplicial vertices efficiently. We formally define simplicial vertices below.

Let $N(v)$ be the set $\{w | (v, w) \in E\}$ and $N[v] = N(v) \cup \{v\}$. That is, $N(v)$ is the neighbourhood of the vertex v and $N[v]$ is the closed neighbourhood of v. We denote

the degree of the vertex defined as $|N(v)|$ by $d(v)$.

Definition 3.1 A vertex v in a graph G is called a simplicial vertex if it's neighbourhood $N(v)$ is complete. ∎

Lemma 3.1 *A vertex v is simplicial if and only if for all neighbours w of v, $N[v] \subseteq N[w]$.*

Proof: For the 'if' part let v be simplicial vertex and $N[v] \not\subseteq N[w]$ for some $w \in V$ and $(v, w) \in E$. Then, it should be the case that there is a vertex u such that $(v, u) \in E$ and $(u, w) \notin E$. Thereby, by Definition 3.1, v is not a simplicial vertex resulting in a contradiction.
For the 'only if' part, it can be observed that as for all neighbours w of v $N[v] \subseteq N[w]$ the neighbours of v must form a complete subgraph.
Hence the proof. ∎

To obtain a parallel algorithm for listing all simplicial vertices we proceed as follows. A sequential algorithm for this problem appears in [37]. Let A be the adjacency matrix of the given graph with diagonal entries set to 1. Find the square of the adjacency matrix, A^2. The following observation holds.

Observation 3.1

$$A^2[u, v] = |N[u] \cap N[v]|$$

∎

Also, from Lemma 3.1 for all simplicial vertices u of G we have

Observation 3.2

$$|N[u] \cap N[v]| = |N[u]|$$

∎

From the Observations 3.1.3.2 the algorithm is complete after the following step. If $A^2(u, v) = A^2(u, u)$ for all neighbours v of a vertex u then u is a simplicial vertex. The work for this part is $O(m)$ at $d(v)$ processors per each vertex. Thus the total cost id $O(n^\alpha)$. The time taken would be $O(\log n)$. The following theorem follows.

Theorem 3.3 *The set of all simplicial vertices in a given graph can be obtained in $O(\log n)$ parallel time using $O(n^\alpha)$ processors on the CREW model of computation.*

The above algorithm can be improved as follows. Let Δ be a parameter to be chosen later. Classify the vertices of the graph as low degree or high degree vertices depending on whether $d(v) < \Delta$ or $d(v) > \Delta$. Let L be the set of low degree vertices and H be the set of high degree vertices. The following observation holds.

Observation 3.3 *If a vertex v of high degree is simplicial then all its neighbours are of high degree.*

Proof: If on the contrary, v is a high degree simplicial vertex and has a low degree neighbour w, then by Lemma 3.1 $|N[w] \geq |N[v]| = d(v) + 1 \geq \Delta + 1$. Thus w would also be classified as a high degree vertex. ∎

Thus we obtain the given below.

Algorithm List-Simplicial-Vertices()
begin
Step 1: Classify vertices into low degree and high degree vertices;
Step 2: Search all simplicial vertices of low degree by explicit checking;
Step 3: In the graph G_H mark all vertices that are incident to a low degree neighbour;
Step 4: Search al simplicial vertices of high degree using matrix multiplication;
end.{Algorithm}

The above algorithm can be implemented in parallel $O(\log n)$ time. The cost is given by $O(m\Delta + (\frac{2m}{\Delta})^\alpha)$ as Step 2 takes $O(\sum_{v \in L} d(v)^2) \leq 2\Delta m$ and there are atmost $\frac{2m}{\Delta}$ vertices of H and matrix multiplication costs $O((\frac{2m}{\Delta})^\alpha)$. At the value of $\Delta = m^{\frac{\alpha-1}{\alpha+1}}$ the cost of the entire algorithm is $O(m^{\frac{3\alpha}{\alpha+1}})$.

Alternatively, a constant time parallel algorithm for this problem can be given along the lines of Section 3.2. The idea is to assign to each vertex $O(d(v)^2)$ processors which check the adjacency of each pair of neighbours of that vertex. Thus the number of processors used would be $\sum_{v \in V} d(v)^2$ which can be bounded as in Section 3.2.

Chapter 4

On the Parallel Complexity of Maximal Matching

Presently, optimal parallel algorithms for computing a maximal matching in a general graph as well as bipartite graph take same time. In the sequential case for the unweighted maximum matching problem the best known algorithms for general graphs and bipartite graphs take the same time $O(m\sqrt{n})$. But the algorithm for bipartite graphs is based on network flow techniques and is much simpler than the algorithm for general graphs which is based on a theory of alternating paths and blossoms.

So, there is evidence to conjecture the existence of a simpler and/or faster optimal parallel algorithm for maximal matching in bipartite graphs exists. We show that such an algorithm results in a improvement in the case of general graphs also.

4.1 Introduction

A matching in a graph G is a subset of edges M such that no two of them share a common end point. A matching M is maximal if M is not a proper subset of another matching. In this Chapter we relate the complexity of computing a maximal matching in a graph to the complexity of computing a maximal matching in a bipartite graph.

The parallel computation of maximal matching is a well studied problem. Israeli and Shiloach [28] gave an randomized $O(\log n)$ time algorithm and a deterministic

$O(\log^3 n)$ time algorithm with $O(m+n)$ processors. Han [26] showed that the problem has an $O(\log^{2.5} n)$ time algorithm with $O((m+n)/\log^{0.5} n)$ processors. This algorithm is suboptimal. The cost of the algorithm is $O((m+n)\log^2 n)$. Han [26] also gives an $O(\log^2 n)$ time $O(M(n))$ processor algorithm where $M(n)$ is the processors required to multiply two $n \times n$ matrices. Currently $M(n) \approx n^{2.4}$. Kelsen [35] gave a $O(\log^4 n)$ time optimal algorithm for general graphs and $O(\log^3 n)$ time optimal algorithm for bipartite graphs. This was later improved by Han [25] to $O(\log^3 n)$ optimal algorithm for general graphs.

4.2 Reduction

We give a decomposition scheme for decomposition of a given graph $G = (V, E)$ into a set of edge disjoint bipartite graphs. The result is a generalization of the result of Kelsen [35].

Lemma 4.1 *Any graph $G = (V, E)$ can be decomposed in to $O(\log C_v(n, m))$ edge disjoint bipartite graphs if the graph G is validly vertex coloured using $C_v(n, m)$ colours.*

Proof: Let a vertex colouring algorithm colour the vertices of the graph using $C - v(n, m)$ colours. Each colour can be interpreted as a bit string of length $O(\log C_v(n, m))$. We refer to the colour of vertex v as $C_v(v)$. $S_v(v)$ may refer to the bit string corresponding to $C_v(v)$. We now partition the edge set such that the edge $e = (u, v)$ is kept in the subgraph $i = \min\{j | j^{th}$ bit of $S_v(u) \oplus S_v(v) = 1\}$. With this we can obtain a $O(\log C_v(n, m))$ bipartite decomposition of G. ∎

Such a decomposition as a bipartite decomposition of G and the number of subgraphs obtained is called the size of the decomposition [35]. Our reduction is according to the following algorithm.

Step 1: Number the vertices of G from 0 to $n-1$.

Step 2: Partition the edges such that $e = (u, v)$ is kept in the graph G_i iff $i = \min\{j | jth$ bit of $u \oplus v = 1\}$.

comment: this is to obtain the bipartite decomposition of size $O(\log n)$

Step 3: Run prefix sums on each G_i to count the size of each G_i.

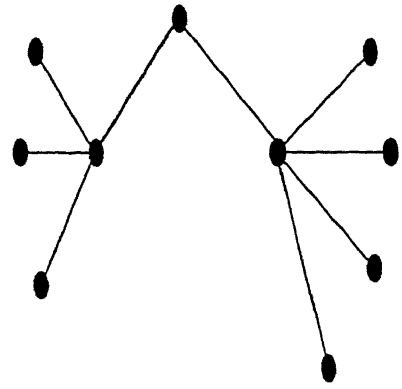


Figure 10: Merge does not give a maximal matching

Step 4. Use optimal algorithm for bipartite graphs to find a maximal matching in each G_i in parallel.
Step 5. Obtain a set of matching edges by colouring the graph using $O(\log n)$ colours with all edges of maximal matching from subgraph i coloured with i.
Step 6. Merge the maximal matchings of the subgraphs pairwise

Kelsen [35] has shown that Steps 1-3 can be done in $O(\log n)$ optimal time. The merging process of Step 6 is described in detail below. Let G_i and G_j be any two subgraphs and M_i and M_j be the maximal matchings of these subgraphs. In $M_i \cup M_j$, each vertex has atmost 2 edges incident at it and one is coloured red and the other blue. Since the graph is of bounded degree, the computation of maximal matching is optimal and within $O(\log n)$ time. The graph can be decomposed into list graphs in constant time. Now list ranking can be used to 2-color the edges of the graph. Let the colours used be c_1 and c_2. Take edges of colour c_1 and obtain a maximal matching. The time is $O(\log n)$ optimal.

But the above algorithm does not gaurentee that the set of edges so obtained is a maximal matching of the original graph. This is so because during Step 6, some vertices which were matched in the subgraphs earlier may now become free. There can be some free neighbours for these vertices and the matching may fail to be maximal. (See Figure 10.) Thus we need to enlarge the matching we have obtained so far to get a maximal matching. For this we proceed as follows. Let the set of edges in the matching obtained be M. Remove all edges which share an end point with some

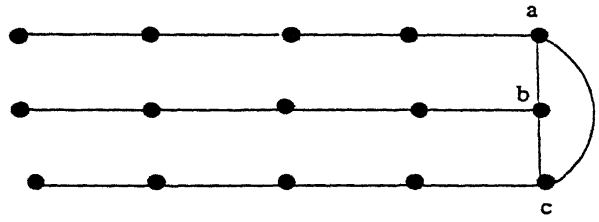


Figure 11: The bad case. The vertices a, b, c will not be covered by the matching

edge in M and remove all isolated vertices. Let F be the set of vertices that are free originally, that is before merging. Let S be the set of vertices that became free due to merging. Consider the graph $G' = (F \cup S, E')$ where E' consists of the edges of the original graph between the vertices of F and S. This graph is bipartite by construction with F and S being the bipartition. Run bipartite maximal matching algorithm on this graph. Update M as $M \cup M'$ where M' is the maximal matching returned by the bipartite maximal matching algorithm for the graph G'.

Step 7. Obtain a bipartite graph with the bipartition (F, S) and use bipartite maximal matching algorithm to obtain a maximal matching

But even this step does not ensure that the maximal matching is computed. The matching may fail to cover vertices of S that do not have any *free* neighbours. (See Figure 11.)

As the number of such vertices can be as high as the number of the even paths, the number of vertices in the auxiliary graph can be about a constant fraction of the number of vertices of G. The number of edges also reduce by a constant factor. So we use the above scheme for $2 \log\log n$ iterations. As problem size reduces to $O(n/c^{2 \log\log n})$ for some constant c, we use the nonoptimal algorithm. The time required will be $t_{so}(n, m)$. But the cost reduces to $O(\frac{m+n}{\log^2 n} \log^2 n) = O(m+n)$ as the cost of the nonoptimal algorithm is $O((m+n)\log^2 n)$.

Step 8. Use the nonoptimal algorithm to compute the maximal matching in the remaining graph

We now analyze the time for the entire schema. The computation of M takes time $t_b(n, m) + \log n$. The computation of maximal matching in G' takes again time

$t_b(n,m)$. This is repeated for $2\log\log n$ iterations. We need another $t_{so}(n,m)$time to obtain the maximal matching of the original graph. Thus the total time is $O(t_b(n,m)\log\log n + t_{so}(n,m))$. Hence the following theorem follows.

Theorem 4.1 *If $t_b(n,m)$is the time taken for computing a maximal matching in a bipartite graph and there is an algorithm $\mathcal{A}$ to compute a maximal matching in a general graph which is polylogarithmic away from optimality then there is an optimal algorithm that runs in time $O(t_b(n,m)\log\log n + t_{so}(n,m))$.* ∎

4.3 Final Remarks

From Theorem 4.1, we have obtained an equation relating the parallel complexity of maximal matching in a bipartite graph and a general graph. Currently, $t_b(n,m)$= $O(\log^3 n)$ [35] and $t_{so}(n,m)$= $O(\log^{2.5} n)$ [26] and is nonoptimal. Also, Han [25] gave an optimal $O(\log^3 n)$ algorithm. This, we believe is rare as the time complexity of bipartite maximal matching is same as that of general graph. So any improvement in the complexity of the bipartite maximal matching problem can be made to reflect in improvement in the time complexity of the general graph.

Also, we can tighten Theorem 2 of Kelsen [35] as follows. The following theorem is from Kelsen [35].

Theorem 4.2 **([35, Theorem 2])** *If there is a parallel algorithm that computes a maximal matching in bipartite graphs in time $t(n) = \Omega(\log n)$, then there is a parallel algorithm that computes a maximal matching in a general graph in time $O(t(n)\log n)$. If the first algorithm is optimal so is the second.*

Using Lemma 4.1, we can tighten Theorem 4.2 and replace the term $O(t(n)\log n)$ by $O(t(n)\log|C(v)|)$ where $|C(v)|$ is the number of colours that the vertex colouring algorithm uses. The only requirement we insist on the vertex colouring algorithm is that it should be optimal. This removes our dependence on the nonoptimal algorithm for maximal matching from Theorem 4.1.

Chapter 5

Parallel Edge Colouring of Graphs

Linial [40] has given a nonconstructive distributed algorithm to find an $5\Delta^2 \log n$ colouring in one round and had left the problem of constructing explicit decomposition open. This has been solved in this chapter by giving an $\Delta^2 \log n$ edge colouring in constant time. Further, if we are provided with a vertex colouring using k colours we show how to reduce the number of colours to $O(\Delta^2 \log k)$. This was also stated in Linial [40] but the proofs are nonconstructive.

5.1 Introduction

Graph colouring, vertex colouring as well as edge colouring, is amongst the most extensively studied problem in graph theory because of its applications in scheduling, optimization and other related problems in operations research. An example of application of edge colouring is the file transfer problem in computer networks. This problem was introduced by Coffman *et al.* [11]. In this problem, every node in the network has a communication port and there are several files to be transfered in between the pairs of nodes. The problem is to schedule the file transfers so as to minimize the total time taken for all the transfers.

In the edge colouring problem we seek to assign colours to edges of the graph such that no two edges sharing a common end point get the same colour. The minimum number of colours for which such a colouring exists is called the edge chromatic

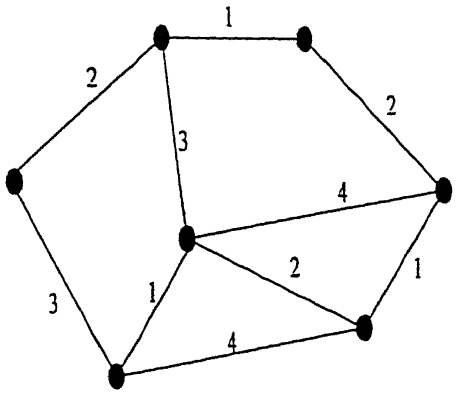


Figure 12: A 4-edge colouring of graph G. The numbers at each edge represent the colour assigned to that edge.

number of the graph and is denoted by $\chi'(G)$. An example of edge colouring is shown in Figure 12 using 4 colours. Clearly, $\chi'(G) \geq \Delta$ where Δ is the maximum degree of any vertex of G. Vizing [31, 19, 42] and Gupta [42] showed that $\chi' \leq \Delta + 1$. This is popularly known as Vizing's theorem. Thus any graph can be edge coloured using either Δ or $\Delta + 1$ colours. Graphs with $\chi'(G) = \Delta$ are called as Class 1 graphs and graphs with $\chi'(G) = \Delta + 1$ are called Class 2 graphs [19]. The problem of deciding whether a graph belongs to Class 1 or Class 2 is called the classification problem of graphs [19]. This problem is shown to be NP-Complete even when restricted to cubic graphs ($\Delta = 3$) by Holyer [27]. Thus edge colouring of graphs with optimal number of colours is computationally difficult.

Approaches like restricting to a particular type of graphs to get optimal colouring or to settle for a near optimal edge colouring are followed and have yielded good algorithms on the sequential model of computation. On the parallel model of computation, namely PRAM(described briefly later in Section 5.2), NC algorithms for edge colouring planar graphs and bipartite graphs exist [10, 9, 38]. But $\Delta + 1$ edge colouring arbitrary graphs is not known to be in NC. Existing algorithms [34, 39] have time requirements in $O(\Delta^k polylog(n))$ for $k > 0$ and hence belong to NC only if the maximum degree of the graph is of the order of $(\log n)^k$ for some constant k. Otherwise they take time proportional to $O((\Delta)^k)$. Furer and Raghavachari [23] gave an Δ^2 approximate edge colouring algorithm for arbitrary graphs in $O(\log^* n)$ time.

In this chapter we give an $\Delta^2 \log n$ edge colouring algorithm. We also show that we

can reduce the number of colours to if we are provided with a valid vertex colouring.

5.2 Definitions

An edge colouring of a graph is a mapping from $C : E \rightarrow N$. An edge colouring is said to be valid iff no two edges in the graph incident at a same vertex get the same colour. A colouring that uses $k = |C|$ colours and is valid is said to be a $k-$colouring of the given graph. The minimum number of colours required to validly edge colour the given graph is called the edge-chromatic number or the chromatic index of the graph and is denoted by χ'. A colouring is said to be minimal if it uses the least possible number of colours over all valid colourings. Any χ' edge colouring of G is called an optimal edge colouring. For sake of convenience, we represent colours by integers.

5.3 Edge Colouring Graphs in $O(1)$ Time

In this section, we consider the edge colouring of general graphs. As was mentioned in Section 5.1, the problem of $\Delta + 1$ edge colouring general graphs is not known to be in NC. Many graph problems that did not have efficient parallel algorithms are solved efficiently when restricted to bipartite graphs [35, 25]. We employ this approach to reduce the edge colouring problem of arbitrary graphs to edge colouring bipartite graphs at the cost of extra colours.

We make use of the generalized bipartite decomposition of a given graph into edge bipartite graphs introduced in Chapter 4. We recall it in the following Lemma.

Lemma 5.1 *Any graph $G = (V, E)$ can be decomposed in to $O(\log C_v(n, m))$ edge disjoint bipartite graphs if the graph G is validly vertex coloured using $C_v(n, m)$ colours.*

Proof: Proof is given in Chapter 4. ∎

We now state a result regarding edge colouring bipartite graphs. The following Theorem is from Furer and Raghavachari [23].

Theorem 5.1 *A bipartite graph G can be validly edge coloured using $O(\Delta^2)$ colours in constant time using $O(n+m)$ processors.*

Proof: The proof is based on the fact that if we have a bipartition of the graph G, then each edge can be treated as a half edge and every vertex colours the half edges incident at it using a palatte of size Δ_v. Combining the colours of each half edge gives a valid $O(\Delta^2)$ edge colouring of G. ∎

We now describe our algorithm for obtaining a constant time algorithm for edge colouring of graphs.

Algorithm Edge-Colour-Graph(G)
Input: A graph $G = (V, E)$
Output: A $O(\Delta^2 \log n)$ valid edge colouring of G
begin
Step 1. Obtain a $O(\log n)$ decomposition of G as given below
Step 1.1. number the vertices from 0 to $n-1$
Step 1.2. place edge $e = (u, v)$ in subgraph i if $i = min\{j|$ bit j of $u \oplus v = 1\}$
comment: Each G_i, $i = 0, 1, \cdots, \log n$ is bipartite from the construction .
Step 2. Colour each subgraph in constant time using Theorem 5.1
comment: Each subgraph can be coloured using Δ^2 colours where Δ is the maximum degree of G.
end.{Algorithm}

The constant time follows because we do not run prefix sums after the decomposition. Instead, we assign a processor to each edge and to every vertex. The computation at each edge is done with the processor assigned to it. As the maximum degree in each subgraph is atmost Δ, the number of colours used would be,

$$\begin{aligned} N(n,m) &\leq \sum_{i=0}^{\log n} \Delta^2 \\ &\leq \Delta^2 \log n \end{aligned}$$

Thus we give an explicit algorithm to edge colour a graph in constant time using atmost $\Delta^2 \log n$ colours.

In the following we assume that we are provided with k valid vertex colouring algorithm and reduce the number of colours required by the edge colouring algorithm. After having a k vertex colouring, we can get a $O(\log k)$ bipartite decomposition of G (From Theorem 4.1.) Thus the number of colours used would be

$$\begin{aligned} N(n,m) &\leq \sum_{i=0}^{\log k} \Delta^2 \\ &\leq \Delta^2 \log k \end{aligned}$$

From the above equation, edge colouring of graphs is equivalent to vertex colouring at the cost of extra colours.

5.4 Edge Colouring of General Graphs

In this section we give an algorithm for edge colouring general graphs. The algorithm uses edge colouring algorithms for bipartite graphs. We assume that there is an algorithm that can colour bipartite graphs in $O(t_b(n,m))$ time using $c_b(n,m,\Delta)$ colours. The algorithm is assumed to employ a linear number of processors. The algorithm for general graphs follows.

Algorithm Edge-Colour-General-Graph(G)
Input: A graph $G = (V, E)$ in adjacency list representation
Output: A valid edge colouring of G.
begin
  Step 1. Obtain a $O(\log n)$ bipartite decomposition of G
  comment: This takes $O(\log n)$ time and $O(m+n)$ work [35].
  Step 2. Colour each bipartite graph independently in parallel using $c_b(n,m)$ colours.
  Step 3. Combine the colours to produce a valid colouring of G.
end. { Algorithm }

The number of colours used and the time requirements are given in the following Lemmas.

Lemma 5.2 *In the $O(\log n)$ decomposition of G, the degree of subgraph i is atmost $min\{2^i, \Delta\}$ where Δ is the maximum degree of G.*

Proof: The proof follows from the observation that there can be atmost 2^i vertices that differ in LSB position i. ∎

Lemma 5.3 *The time taken by the above algorithm is $O(\log n + t_b(n, m))$.*

Proof: The time for obtaining $O(\log n)$ bipartite decomposition is $O(\log n)$ using $O(m+n)$ work [35]. Step 2 takes $O(t_b(n, m))$ time. Hence the total time is $O(\log n + t_b(n, m))$. ∎

Lemma 5.4 *The algorithm uses $c_b(n, m, 2^i) \log \Delta + c_b(n, m, \Delta) \log(\frac{n}{\Delta})$ colours.*

Proof: By Lemma 5.2 the degree of each subgraph is atmost $\min\{2^i, \Delta\}$. Hence if we assume that the bipartite graph edge colouring algorithm uses $c_b(n, m, \Delta)$ colours, the number of colours used for the general graphs would be

$$\begin{aligned} N(n, m) &= \sum_{i=0}^{\log n} c_b(n, m, min\{2^i, \Delta\}) \\ &= \sum_{i=0}^{\log \Delta} c_b(n, m, 2^i) + \sum_{i=\log \Delta}^{\log n} c_b(n, m, \Delta) \\ &= c_b(n, m, 2^i) \log \Delta + c_b(n, m, \Delta) \log(\frac{n}{\Delta}) \end{aligned}$$

∎

5.5 Conclusions

In this Chapter, we have given constructive proofs for two results stated by Linial [40]. Further, the constructions and algorithms are very simple.

Chapter 6

Minima in Sublinear Time

6.1 Introduction

The problems involving integers drawn from a restricted universe have received considerable attention. Integer sorting [2, 4, 36], for example, is very well studied and better algorithms in sequential as well as parallel settings are still sought. Data structures that support priority queue operations in less than $O(\log n)$ time per operation are also devised [15], [16]. In parallel setting for related problems like integer prefix computation good lower bounds are also proved by Chaudari and Radhakrishnan [8]. The bound proved is $O(\alpha(n))$ using a linear number of processors.

In this Chapter we consider the problem of finding the minima of n integers $x_1, x_2, \cdots, x_n$ from the universe $U = [0 \cdots m-1]$. Although the information theoretic bound of $O(n)$ is established the universality of the bound differs greatly. By enhancing the machine instruction set to include bit-wise operations like shifting and logical operations like AND, OR etc. it is shown that for problems like integer sorting it is possible to beat the information theoretic lower bound [22].

In this Chapter we show that using a machine model that provides support for bit-wise operations and logical operations, we can cross the lower bound of $O(n)$ for the problem of finding the minimum of n integers. The bound established is $O(n/\log n)$ in the sequential case. The main idea we use namely packing of integers in a single machine word is excessively used. In integer sorting of [2] a variant of

bitonic sorting is used and the ability to pack integers results in a better time than $O(n \log n)$. Similar techniques were also used in [4, 22].

Kirkpatrick [36] introduces the notion of conservative and nonconservative nature of the algorithms. An algorithm is said to be conservative if it uses a word size of $O(\log m + \log n)$ for n integers taken from the universe $U = [0 \cdots m-1]$. Otherwise the algorithm is nonconservative. Our algorithm is nonconservative. Although integer sorting problem is well studied on the conservative as well as the nonconservative models of RAM's and PRAM's not much is known about the bounds of other problems on the nonconservative PRAM wherein we can exploit the word parallelism to obtain better bounds [14].

The rest of the Chapter is organized as follows. In section 6.2 we present a sequential algorithm to compute the minima of n integers in $O(n/\log n)$ time thereby proving our main result. In section 6.3 we give a parallel version of our algorithm that runs in $O(\log n)$ time and having a cost of $O(n/k)$ with word size of $O(k * \log m)$. In Section 6.4 discusses some issues relevant to the performance of the algorithm.

6.2 Sequential Algorithm

We first give the format of the machine and the machine word that we make assume. The machine supports the operations LSHIFT and RSHIFT defined below. LSHIFT, $x = y << z$ means that the value obtained by left-shifting y by z times is assigned to x. Observe that the value z is available as a pre-computed constant or program generated value [22]. It is equivalent to multiplying y by 2^z. The operation RSHIFT, $x = y >> z$ amounts to dividing y by 2^z. Thus our machine instruction set consists of { +, $\dot{-}$, LSHIFT, RSHIFT, AND, NOT } . Although this set appears to be less powerful than the conventional instruction set of { +, -, x, $\lfloor / \rfloor$ }, it is indeed the case that the former introduces the power of exponentiation, if shifts are used in a completely unrestricted fashion, as observed by Kirkpatrick [36]. In our machine a word consists of k integers, for a parameter k, and is of the form shown in figure 13. The word has fields numbered 1 to k from right to left. The extra bits serving as delimiters, which we call as indicator bit can be suitably set or reset as per requirement

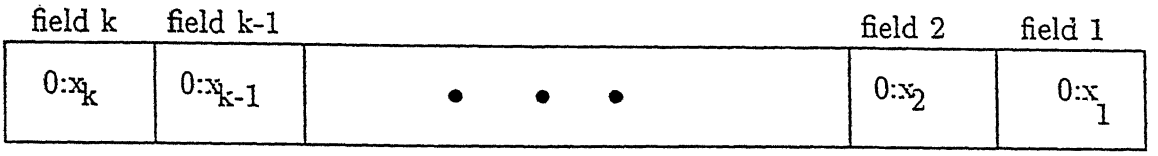


Figure 13: Format of a machine word

and can be done by appropriate AND-ing or OR-ing. Such a word is written as $[0:x_k, 0:x_{k-1}, \cdots, 0:x_1]$.

We now present a simple routine to find the minimum of the integers containec in two machine words in $O(\log k)$ time.

6.2.1 Minima of two words

Our algorithm has a preprocessing phase which is described below. We need th constant word $K1 = [1:0, 1:0, \cdots, 1:0]$. The word $K1$ can be generated using th following code [2]. Let $l = \lceil \log m \rceil$.

$K1 = 1 << l$;
for $i = 1$ to $\log k$ do
    $K1 = (K1 << 2^i * l)$ AND $K1$;
endfor;

Clearly, the above routine takes $O(\log k)$ time.

We also need some masks during the running of the algorithm. Because of tl symmetric nature of the masks it is possible to generate them in constant time on one of them is generated. The masks required are of the form $(1)^r(0)^r$ for valu of r namely $r = k/2, k/4, \cdots$, when the indicator bits are viewed as a string of 1 and $0's$. The field is to contain 0 in all the positions. We generate the mask f $r = k/2$ explicitly. Using that we show how to generate successive masks. The co to generate the mask for $r = k/2$, we call it $M1$, follows.

/* $M1 = (1)^{k/2}(0)^{k/2}$ */
/* Use $K1$ generated earlier */
$M1 = K1 >> 2^{k/2} * l$;

$M1 = K1 << 2^{k/2} * l;$

As can be seen, this routine takes constant time. Moreover, it can be easily observed that the generation of other masks is trivial. This completes the preprocessing step and we now give the algorithm to find the minimum of the integers in two machine words.

procedure minimum_2(A,B)
/* A and B are the two words each containing k integers shown in the format of figure 13. Word A has indicator bits set to 1.
for $t = 1$ to $\log k$ do
/* subtract the words A and B */
$K2 = A$ - B ;
/* Need only indicator bits */
$K2 = K2$ AND $K1$;
/* extract the local minima */
$C1$ = A - ($K2$ AND A);
$C2$ = B AND $K2$;
$C = C1$ OR $C2$;
/* split the word C */
$C1 = C$ AND M_t;
$C2 = C$ - $C1$;
$A = C1 >> 2^t * l$;
$B = C2$;
endfor;
end;

Clearly, each iteration of the above algorithm takes constant time in our machine model and hence the entire algorithm takes $O(\log k)$ time.

6.2.2 Minima of n integers

We now use the above algorithm to compute the minimum of n integers $x_1, x_2, \cdots, x_n$ drawn from the universe $U = [0 \cdots m-1]$. The algorithm follows.

```
procedure minima_n()
    for i = 1 to n/2 * k do
        find the minima of n/2 * k pairs of words
    endfor;
    find the minima of the remaining n/2 * k integers by
    recurring over problem of size O(n/k).
end;
```

The recurrence relation guiding the algorithm is $T(n) \leq n/k * T(k) + T(n/k)$; with $T(n)$ being the time taken to solve the problem of size $O(n)$ and $T(k) = O(\log k)$. The solution for this recurrence can be seen to satisfy $T(n) \leq O(n/k * \log k)$.

Choice of k

Different values of k lead to different algorithms. In particular if $k = O(\log n)$ the time would be $O(n \log\log n / \log n)$ with a word size of $O(\log n \log m)$. But if $k = O(\log\log n)$, the time would reduce to $O(n/\log n)$ but the word size requirement grows to $O(\log n \log\log n * \log m)$.

Theorem 6.1 *The minima of n integers $x_1, x_2, \cdots, x_n$ drawn from the range $U = [0 \cdots m]$ can be computed in time $O(n * \log k/k)$ on a sequential machine using a word size of $O(k * \log m)$, for $k > 1$.*

6.3 Parallel Algorithm

In this section we give a parallel algorithm for the algorithm presented in section 6.2. The model of parallel computation we use is the EREW-PRAM wherein only one processor is allowed to either read or write to the same memory location. A parallel algorithm is judged by two parameters namely the time taken by the algorithm and

the amount of work it does. The amount of work is measured as the product of the time taken and number of processors employed. The information theoretic lower bound established for this problem is $O(\log n)$ time and $O(n)$ work [30]. Our algorithm runs in time $O(\log n)$ time but does only $O(n \log n/k)$ work. We use the algorithm to find the minima of the integers contained in two words given in section 6.2.1. The algorithm follows.

```
procedure minima_n(n)
    for n/2 * k pairs of words pardo
        find the minima of the words in O(log k) time
    endfor
    recur over problem size O(n/k)
end
```

The recurrence relation guiding the time taken by the algorithm is $T(n) = O(\log k) + T(n/k)$ whose solution is $T(n) = O(\log n)$. The number of processors used is $O(n/k)$ and hence the work done is clearly $O(n \log n/k)$.

Theorem 6.2 *The minima of n integers drawn from the range $U = [0 \cdots m-1]$ can be found in time $O(\log n)$ with $O(n/k)$ processors on the EREW-PRAM on a* non-conservative *PRAM.*

6.4 Practical Issues

Our main result that there is a sequential algorithm to compute the minima of n integers drawn from the range $U = [0 \cdots m-1]$ ignores the fact that under any model of computation. it requires $\Omega(n)$ time to read the input of n integers. But we can circumvent this problem if we assume that the input is instead $O(n/k)$ words each containing k integers. This case is not too abnormal as minimum finding is a problem that appears as a subproblem in many other problems and thus we need not charge the time of $\Omega(n)$ for inputting on our algorithm. Rather we can assume that

the superproblem indeed reads the integers one by one taking $O(n)$ time units but as part of computation needs our minimum finding algorithm as a subtask.

In effect, we assume that the input to our algorithm consists of n/k words each containing k integers. Although this may appear out of context and exotic, as Albers and Hagerup [2] note "algorithms using a nonstandard word length should not hastily be discarded as infeasible and beyond practical relevance". We seek to find applications of our algorithm or obtaining better algorithms for related problems which subsequently result in better algorithms for what we call *superproblems* like integer sorting and integer priority queue.

Chapter 7

Conclusions

7.1 Conclusions

In this thesis, we proposed improved algorithms in the following cases. For the problem of minimum spanning tree considered in chapter 2, we have improved the existing result of Pawagi and Ramakrishanan for the edge insertion problem from $O(\log^2 n)$ time $O(n^2)$ work to $O(\log n)$ time and $O(n)$ work. We have also applied the bounded incremental analysis of Ramalingam [44] and obtained an $O(\delta)$ time sequential algorithm for the same problem.

As to our knowledge, we know no existing parallel algorithms for the small subgraph recognition problems. There are some sequential algorithms for this problem [37]. For the claw free graph recognition problem, a sequential simulation of our parallel algorithm improves the result of Kloks *et al.*[37]. Moreover, we are able to obtain constant time for some of the problems thereby getting algorithms that are *fully parallelizable.*

For the maximal matching problem, Theorem 4.1 improves the result of Kelsen [35] to a factor of $\log\log n$. (Theorem 2 of Kelsen [35] proves that the complexity of maximal matching in general graphs is $O(\log n)$ times the complexity of the problem on bipartite graphs.) Also, our schema is optimal as that of Kelsen [35] and Han[25]. Any improvement in the existing algorithms for bipartite graphs will now result in a improvement for the general graphs as well.

For the edge colouring problem we gave a very fast algorithm to produce a valid $\Delta^2 \log n)$ colouring. We also showed how to reduce the number of colours by making use of a generalized decomposition of graphs.

For problems involving integers, integer sorting and priority queues are well studied. We considered the problem of minima of integers and showed that minima can be obtained in less than linear time in certain circumstances.

7.2 Further Work

In this thesis, we have proposed an bounded incremental algorithm for updating the minimum spanning tree under edge insertions. A related problem is to update the minimum spanning tree under multiple edge insertions [17]. That is, to the existing graph, a set of new edges are being added to the current graph.

Obtaining better bounds on the variance of the degree sequence of a graph is also interesting. This would immediately reflect in improvements of the algorithms of chapter 3 both in the sequential case and also the parallel case. Most notably, improvement would be for the case of presence of triangle, recognition of claw free graphs and listing simplicial vertices.

Better expression to relate the time complexities of maximal matching in bipartite graph and general graph are also of interest. Also the applications of bipartite decomposition can be generalized to hypergraphs as observed by Kelsen [35]. So it is a open question to obtain algorithms for computing a maximal matching in hypergraphs , either general or of constant dimension. A Lemma that can help in such problems is sated below. The Lemma is taken from Lovasz [41].

Lemma 7.1 *A r regular hypergraph can be split into two hypergraphs such that atleast one of them contains atmost* $\frac{m}{2^{r-1}}$ *edges.*

Bibliography

[1] AHO, A. V., HOPCROFT, J., AND ULLMAN, J. D. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] ALBERS, S., AND HAGERUP, T. Improved parallel integer sorting without concurrent writing. In *Proceedings of the* 3^{rd} *Annual ACM Symposium on Discrete Algorithms* (1992), pp. 463–472.

[3] ALON, N., YUSTER, R., AND ZWICK, U. Finding and counting given length cylces. In *In Proceedings of ESA'94* (1994), pp. 354–364. Lecture Notes in Computer Science, Vol. 855.

[4] ANDERSON, A., HAGERUP, T., NILSSON, S., AND RAMAN, R. Sorting in linear time? In *Proceedings of the* 27^{th} *Annual ACM Symposium on Theory of Computing* (1995), pp. 427–436.

[5] ARNBORG, S., PROSKUROWSKI, A., AND CORNEIL, D. G. Minimal Forbidden Minor Charecterization of a Class of Graphs. In *Colloquia Mathematica Societatis János Bolyai* (1988), A. Hajnal, L. Lovász, and V. T. Sós, Eds., pp. 49–62. published by North Holland Mathematical Society, Combinatorics.

[6] BOLLABAS, B. *Random Graphs.* Academic Press, 1985.

[7] BOLLOBAS, B. *Extremal Graph Theory.* Academic Press, 1978.

[8] CHOWDARY, S. P., AND RADHAKRISHNAN, J. The complexity of parallel prefix problems on small domains. *Information and Computation 138*, 1 (1998), pp. 1–22.

[9] CHROBAK, M., AND NISHIZEKI, T. Improved edge coluring algorithm for planar graphs. *Journal of Algorithms 11*, 1 (1990), 102–116.

[10] CHROBAK, M., AND YUNG, M. Fast algorithms for edge colouring planar graphs. *Journal of Algorithms 10*, 1 (1989), 35–51.

[11] COFFMANJR, E. G., GAREY, M. R., JOHNSON, D. S., AND LAPAUGH, A. S. Scheduling file transfers. *SIAM Journal on Computing 14* (1985), pp 743–780.

[12] COLE, R., AND VISHKIN, U. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control 70* (1986), 32–53.

[13] COPPERSMITH, D., AND WINOGRAD. On the asymptotic complexity of matrix multipliction. *SIAM Journal on Computing 11* (1982), pp. 471–482.

[14] DESSMARK, A., AND LINGAS, A. On the power of nonconservative pram. In *Proc. of MFCS '96, Lecture Notes in Computer Science,v. 1113* (1996), pp. 303–311.

[15] EMDE-BOAS, P. V. preserving order in a forest in less than logarthmic time and linear space. *Information Processing Letters 6* (1977), pp 80–82.

[16] EMDE-BOAS, P. V., KASS, R., AND ZIJLSTRA, E. Design and implementation of an efficient priority queue. *Math. Systems Theory 10* (1977), pp 99–127.

[17] EPPSTEIN, D., GALIL, Z., AND ITALIANO, G. F. Dynamic graph algorithms. Available at *http://www.uni–paderborn.de/fachbereich/AG /agmadh/Scripts/GENERAL/dyn–survey.ps.gz.*

[18] EPPSTEIN, D., ITALIANO, G. F., TAMASSIA, R., TARJAN, R. E., AND WESTBROOK, J. Maintenance of a minimum spanning forest in a dynamic planar graph. *Journal of Algorithms 13* (1992), pp. 33–54.

[19] FIORINI, S., AND WILSON, J. *Edge Colourings of Graphs.* Pitman.London, 1977.

[20] FREDERICKSON, G. N. Data structures for on-line updating of minimum spanning trees with applications. *SIAM Journal on Computing 14* (1985), pp. 781–798.

[21] FREDERICKSON, G. N. Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees. In *Proceedings of 32nd IEEE Symposium on Foundations of Computer Science* (1991), pp. 632–641.

[22] FREDMAN, M., AND WILLERS, D. E. Blasting through the information theoretic barrier with fusion tress. In *Proceedings of the* 22^{nd} *Annual ACM Symposium on Theory of Computing* (1990), pp. 1–7. Journal Version in Jl. Comp. Sys. Sci., Vol 47,.

[23] FURER, M., AND RAGHAVACHARI, B. Parallel edge colouring approximation. *Parallel Processing Letters 6*, 3 (1996), 321–329.

[24] GOLUMBIC, M. C. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[25] HAN, Y. An improvement on the parallel computation of maximum matching. *Information Processing Letters 56*, 6 (1995), 343–348.

[26] HAN, Y. A fast derandomization scheme and its applications. *SIAM Journal on Computing 25*, 1 (1996), 52–82.

[27] HOLYER, I. The NP Completeness of edge colouring. *SIAM Journal on Computing 10* (1981), 718–720.

[28] ISRAELI, A., AND SHILOACH, Y. An improved parallel algorithm for maximal matching. *Information Proceesing Letters 22* (1986), 57–60. See also Inform. Proc. Lett. 22(1986), 77-80.

[29] ITAI, A., AND RODEH, M. Finding a minimum circuit in a graph. *SIAM Journal on Computing 7* (1978), pp. 413–423.

[30] JAJA, J. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[31] JENSEN, T. R., AND TOFT, B. *Graph Colouring Problems.* John Wiley & Sons Inc., 1995.

[32] JOHNSON, D. B., AND METAXAS, P. Optimal algorithms for single and multiple vertex update problem of a minimum spanning tree. *Algorithmica 16*, 6 (1996), pp 633–648.

[33] JOHNSON, D. S. The NP-Completeness Column: An Ongoing Guide. *Journal of Algorithms 8* (1987), 285–303.

[34] KARLOFF, H. J., AND SHMOYS, D. B. Efficient parallel algorithms for edge colouring problems. *Journal of Algorithms 8*, 1 (1987), 39–52.

[35] KELSEN, P. An optimal parallel algorithm for maximal matching. *Information Processing Letters 52* (1994), 223–228.

[36] KIRKPATRICK, D., AND REISCH, S. Upperbounds for sorting integers on random access machines. *Theoretical Computer Science 28* (1984), pp 263–276.

[37] KLOKS, T., KRATCSH, D., AND MULLER, H. Finding and counting small induced subgraphs efficiently. In *In Proceedings of* 21^{st} *Intl. Workshop on Graph-Theoretic Computer Science, LNCS V.1017* (1995), M. Nagl, Ed., pp. 14–23.

[38] LEV, G. F., PIPPENGER, N., AND VALIANT, L. G. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. on Computers c-30*, 1 (1981), 93–100.

[39] LIANG, W., SHEN, X., AND HU, Q. Parallel algorithm for edge colouring and edge colouring update problems. *Journal of Parallel and Distributed Computing 32*, 1 (1996), 66–73.

[40] LINIAL, N. Locality in distributed graph algorithms. *SIAM Jl. on Computing 21* (1992), pp. 193 – 201.

[41] LOVASZ, L. *Combimatorial Problems and Exercises.* North Holland Mathematical Society, 1993.

[42] MISRA, J., AND GRIES, D. A constructive proof of vizing's theroem. *Information Processing Letters 41* (1992), 131–133.

[43] PAWAGI, S., AND RAMAKRISHNAN, I. V. An $O(\log n)$ algorithm for parallel update of minimum spanning trees. *Information Processing Letters 22*, 5 (1986), pp 223–229.

[44] RAMALINGAM, G. *Bounded Incremental Computation (LNCS V. 1089).* Springer-Verlag, 1996.

[45] SCHEIBER, B., AND VISHKIN, U. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing 17*, 6 (1988), pp. 1253–1262.

[46] TARJAN, R. E., AND VISHKIN, U. An efficient parallel biconnectivity algorithm. *SIAM Journal On Computing 14*, 4 (1985), pp. 862–874.

[47] TARJAN, R. E., AND YAO, A. C. Storing a sparse table. *Comm. of the Assoc. Comp. Mach. 22*, 11 (1979), pp. 606–611.

(( ) ( ))) ( ) (

a) Unbalanced

( ) ( ) ( ) ( ) ( ) ( ) ( ) (

b) NL = 1

(((((((( ))))))))

c) NL = 8

((( ) ( ))) (((( )))

d) NL = 4

Fig. 4 Examples of Balanced Parenthesis

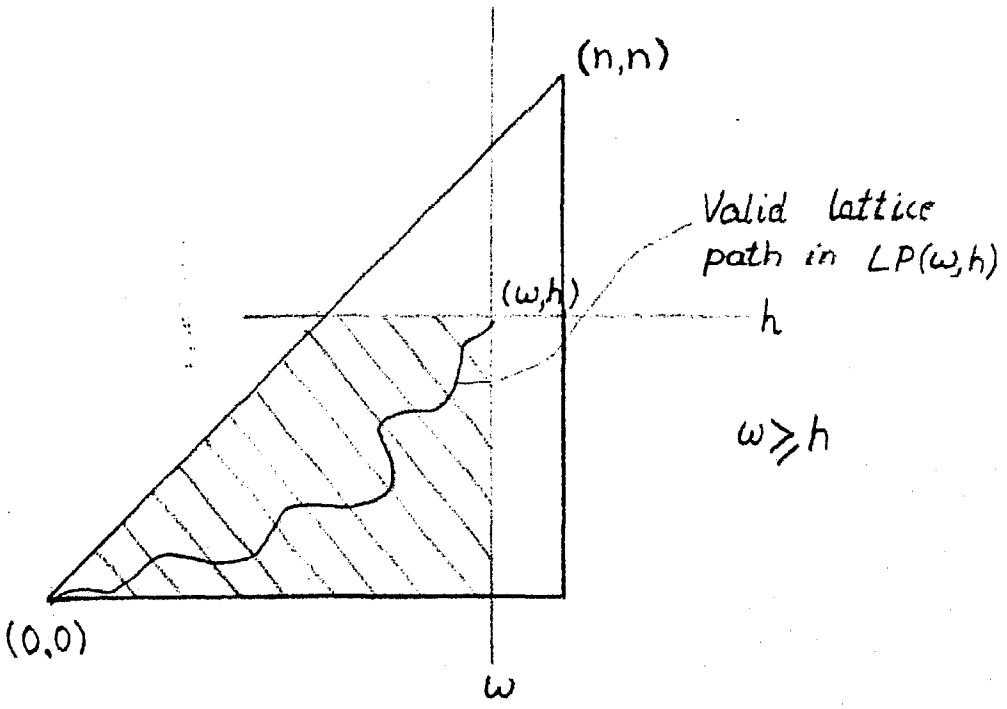


Fig. 5 Problem LP(ω,h)

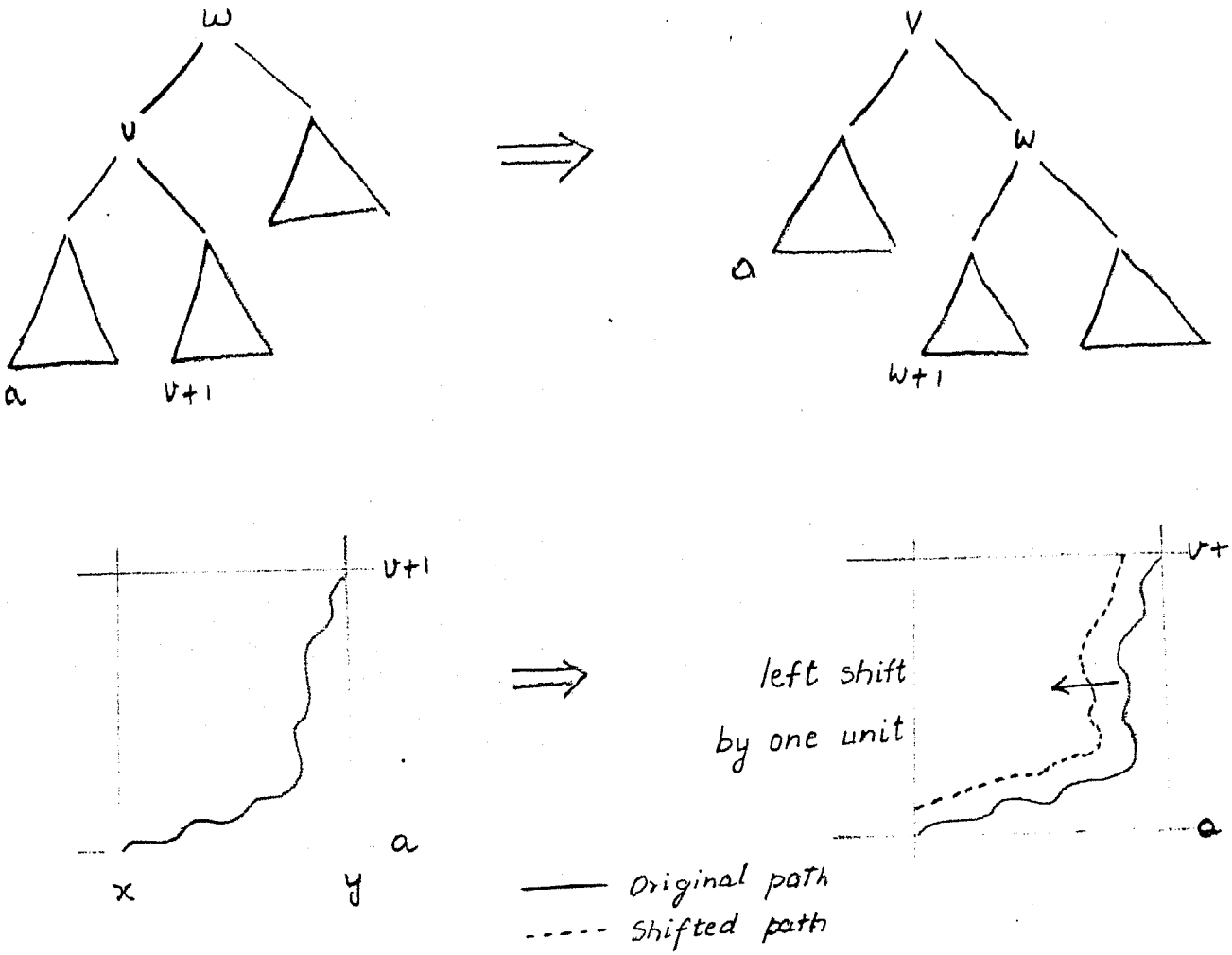
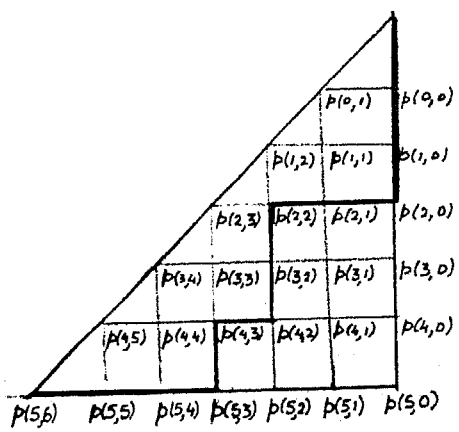


Fig 6. Effect of Tree Rotations on Lattice Paths



rank = $p(5,6) + p(5,5) + p(5,4) + p(4,3) + p(2,$

= 0 + 42 + 42 + 14 + 2 + 2

= 102

Fig 7. Level-Ranking Approach

$$\hat{X} = (3, 1, 0, 2, 0, 0, 0)$$

$$rank_X = p(5,6) + p(5,5) + p(5,4) + p(4,3) + p(2,2) + p(2,1)$$

$$= 0 + 42 + 42 + 14 + 2 + 2$$

$$= 102$$

i	$s1_i$	$s2_i$	$S_i = p(n-s1_i-1, s2_i)$
0	0	6	$p(5,6)$
1	0	5	$p(5,5)$
2	0	4	$p(5,4)$
3	1	3	$p(4,3)$
4	3	2	$p(2,2)$
5	3	1	$p(2,1)$

Fig.8 Parallel Implementation of Level - Ranking

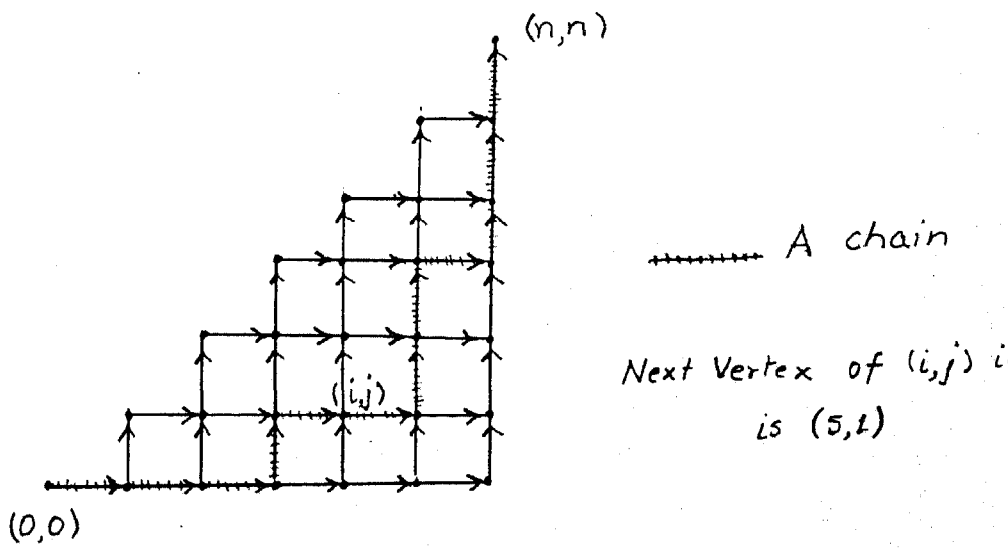
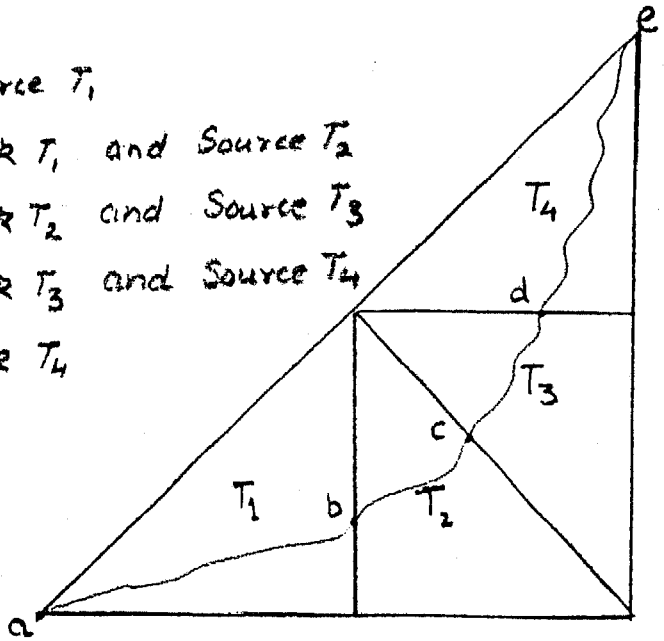


Fig.9 DAG Approach

- Source T_1
- Sink T_1 and Source T_2
- Sink T_2 and Source T_3
- Sink T_3 and Source T_4
  Sink T_4



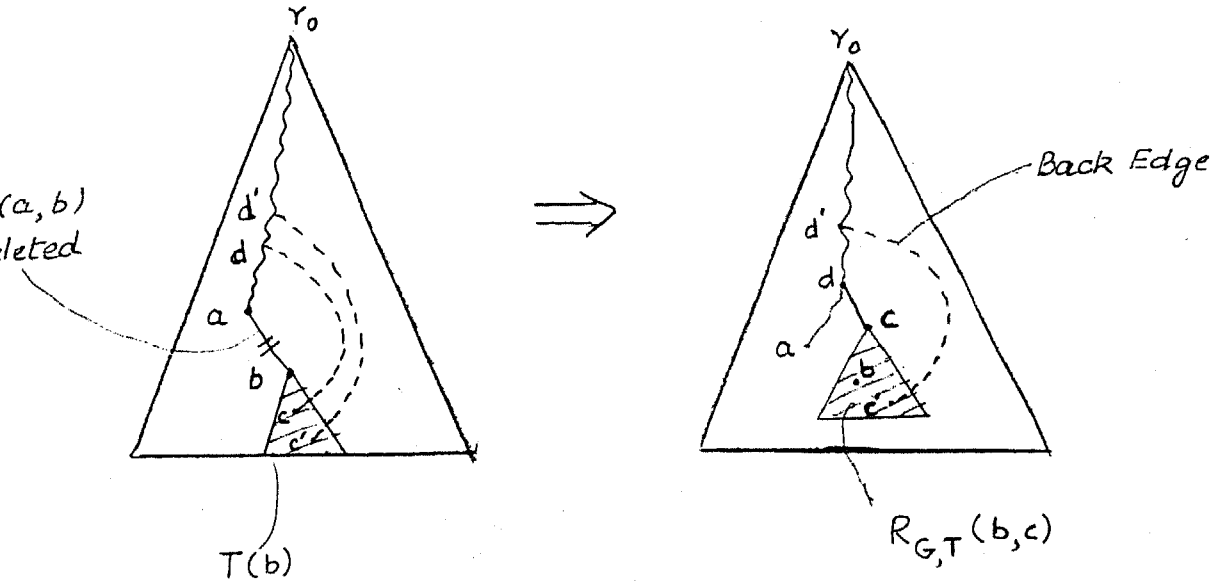
T_1 - Single Source Multiple Sinks

T_2 - Multiple Sources Multiple Sink

T_3 - Multiple Sources Multiple Sink

T_4 - Multiple Sources Single Sin

Fig. 10 Decomposition Approach



Deletion

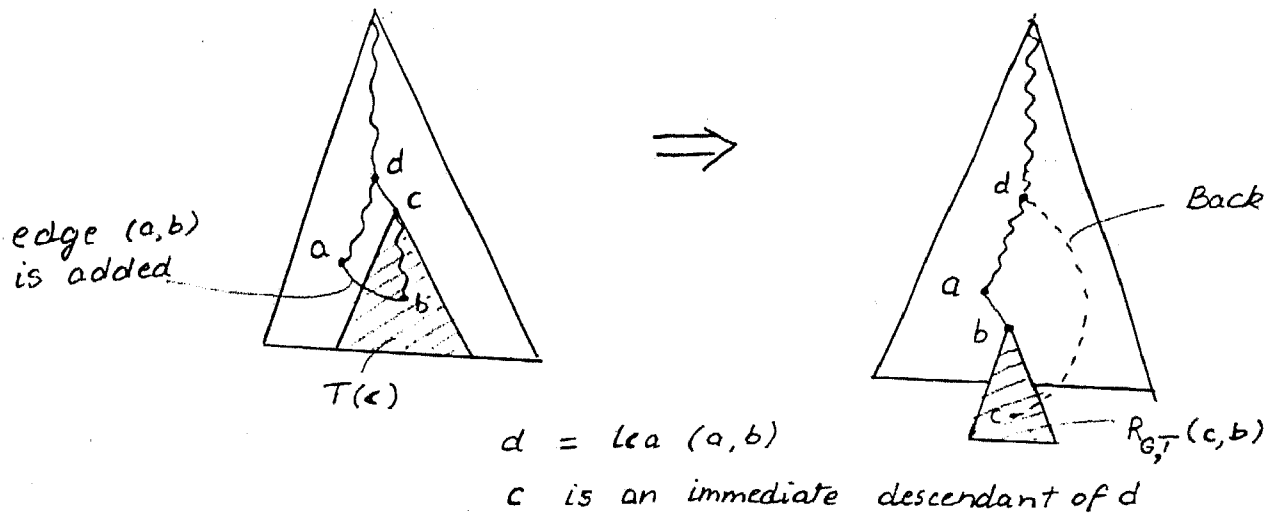


Fig. 12. Edge Addition

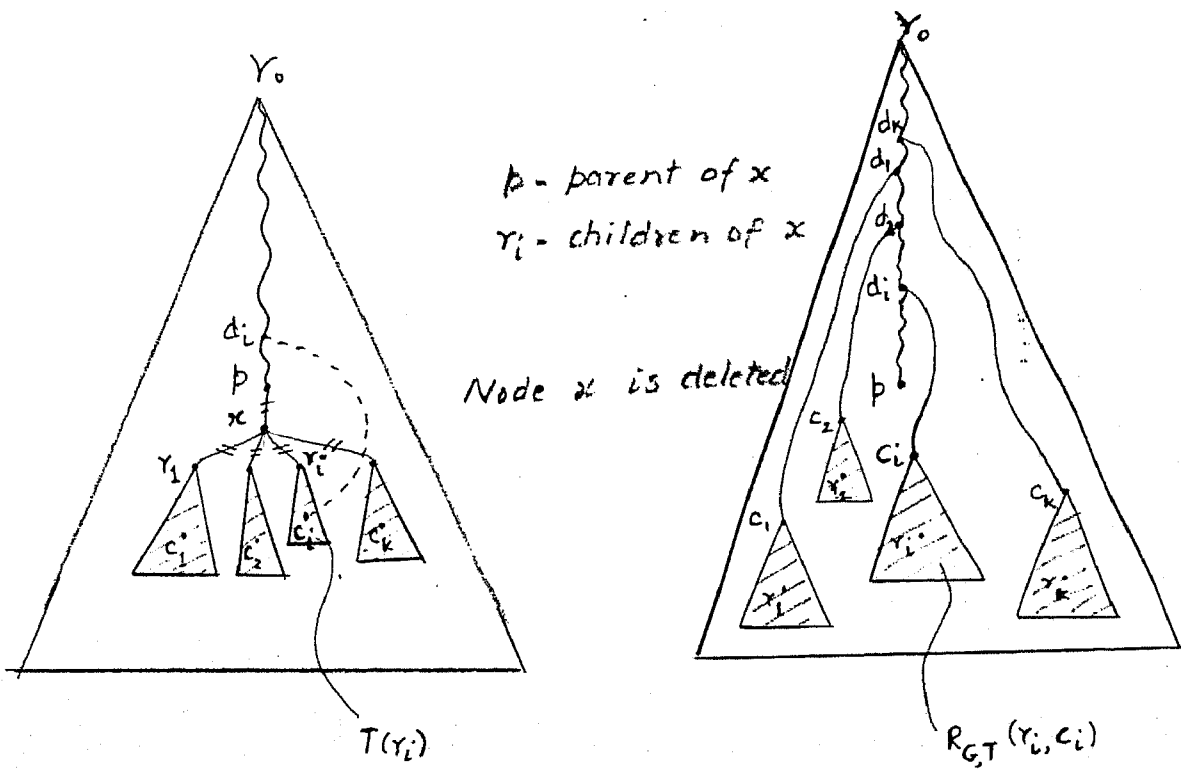
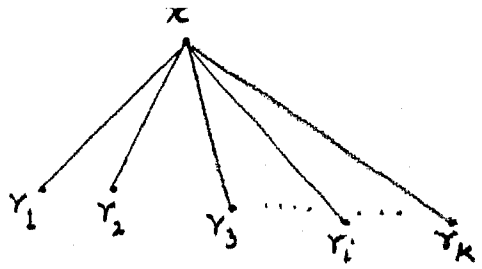


Fig. 13. Vertex Deletion



Vertex x with several incident edges.

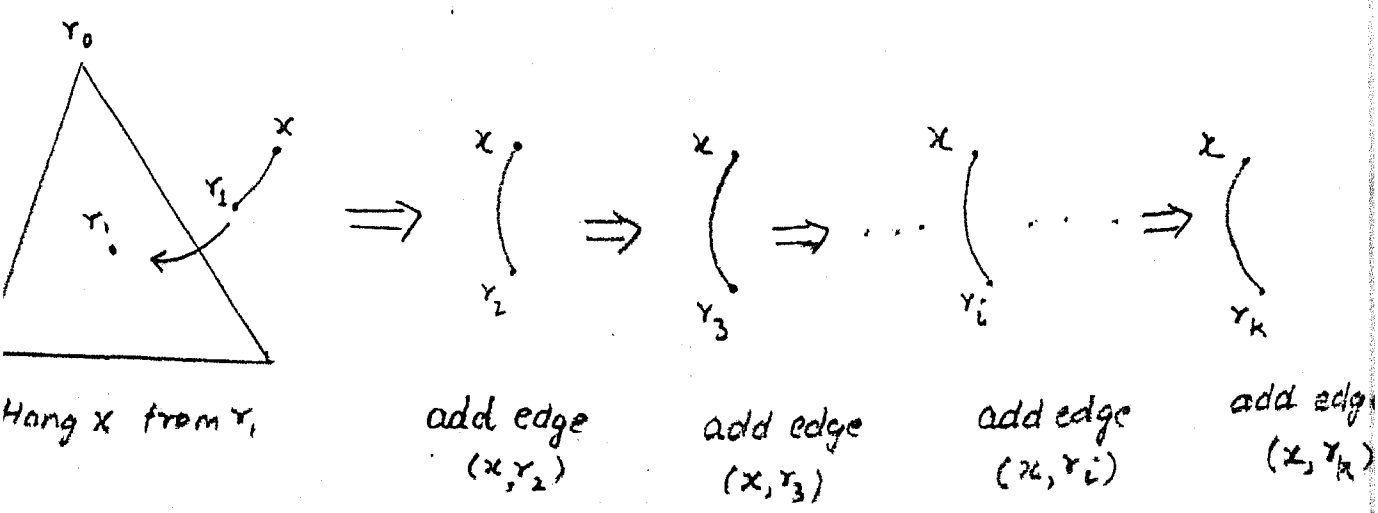


Fig. 14 Vertex Addition